JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 4, June 2007

Formal Validation of Realtime Middleware

John P. Enright*
Ryerson University, 350 Victoria St. Toronto, ONT M5B 2K3
and
David W. Miller†
Massachusetts Institute of Technology, 77 Massachusetts Ave. Cambridge, MA 02139

Distributed terrestrial computer systems employ middleware software to provide communications abstractions and reduce software interface complexity. Embedded applications are adopting the same approaches, but must make provisions to ensure that hard real-time temporal performance can be maintained. This thesis presents the development and validation of a middleware system tailored to spacecraft flight software development. Our middleware runs on the Generalized Flight Operations Processing Simulator (GFLOPS) and is called the GFLOPS Rapid Real-time Development Environment (GRRDE). GRRDE provides publish-subscribe communication services between software components. These services help to reduce the complexity of managing software interfaces. The hard real-time performance of these services has been verified with General Timed Automata modelling demonstrating the usefulness of formal analysis techniques for flight software applications.

Nomenclature

S	Sequence (ordered set), $S = \{s_0, s_1, \ldots, s_{N-1}\} = \{s_n\}\Big|_{n=0}^{N-1}$
s_i	Element of sequence
$head(S)$	Operator: returns first element in sequence
$tail(S)$	Operator: all of S, but the first element. I.e. $\{s_n\}\Big|_{n=1}^{N-1}$
$last(S)$	Operator: returns last element in sequence
$init(S)$	Operator: all of S, but the last element. I.e. $\{s_n\}\Big|_{n=0}^{N-2}$
$S \vdash a$	Operator: appends a to S. I.e. $S \vdash a \equiv \{s_0, s_1, \ldots, s_{N-1}, a\}$
$S \dashv a$	Operator: prepends a to S. I.e. $S \dashv a \equiv \{a, s_0, s_1, \ldots, s_{N-1}\}$
$A\|B$	Operator: concatenates A and B. I.e. $A\|B \equiv \{a_0, a_1, \ldots, a_{M-1}, b_0, b_1, \ldots, b_{N-1}\}$
$a \preceq b$	Predecessor relationship. I.e. $a \preceq b \Leftrightarrow a = s_i, b = s_j, i \leq j$ in some sequence
$A \preceq B$	Subsequence relationship. I.e. $A = \{a_0, a_1, \ldots, a_{M-1}\} = \{b_n\}\Big|_{n=j}^{j+M-1}$

Received 26 February 2007; revision received 23 February 2006; accepted for publication 13 February 2007.

* Assistant Professor, Department of Aerospace Engineering, Member AIAA.
† Professor, Department of Aeronautics and Astronautics, Fellow AIAA.

$A \subseteq B$	Ordered subset. I.e. $\forall(a_i, a_j) \in A, a_i \preceq a_j \Rightarrow \exists(b_n, b_m) \in B, a_i = b_m \wedge a_j = b_n \wedge b_m \preceq b_n$
Δ_{max}	Upper-bound on delay
x	Published value
i	Process index
t	Time
k	Message count

I. Introduction

DESPITE the popularity of various software engineering methodologies, building complex, reliable software systems is still very difficult. That software was directly responsible for the failure of several recent space missions suggests shortcomings of the prevailing development culture.[1] Interconnections between software components are one of the leading causes of errors in complex software. Studies documenting the validation of spacecraft show that interface errors account for 20–35% of all flight software flaws.[2] Clearly, developers must devote much of their attention to the correctness of software connectivity.

Communications complexity can affect flight software on several levels. Concurrent tasks on a single processor must share the available computational resources while communicating effectively with each other. Within a satellite, managing co-operating processors can also be challenging. Coordinating separated spacecraft can be even more problematic, especially if real-time performance is required. When the Iridium communications system came online, users' calls were often dropped, as software attempted to pass control of the call to another satellite.[3] As distribution, either within, or between satellites, becomes commonplace, we expect to see a demand for complexity management in software communications.

In this paper we present an approach to flight-software engineering that addresses the need to mitigate communications complexity, while maintaining a high level of confidence in the safety of the software. Abstract communications services, implemented as middleware, simplifies software design. Formal validation using *General Timed Automata* (*GTA*)[4] allows us to prove safety and performance properties of the system. This balanced approach is both practical and effective.

A. The GFLOPS Testbed

Flight programs are difficult to come by, and not surprisingly, difficult to freely experiment with. Even if we were assigned the task of developing middleware for a particular mission, it is unlikely that we would be afforded the freedom required to explore the range of software concepts valuable to academic research. Instead, we adopt a more modest but versatile application: spacecraft simulation.

This research constitutes part of the *Generalized FLight Operations Processing Simulator (GFLOPS)* program.[5] GFLOPS's goal is to produce a software testbed suitable for high-fidelity, real-time simulation of distributed spacecraft systems. GFLOPS allows the user to develop sophisticated simulations and prototype flight software in a hard real-time environment. During development of the testbed, we identified the need for a software environment that would support rapid application development. High-performance embedded middleware was deemed to be an effective means of achieving this goal.

B. Real-Time Middleware

Middleware commonly refers to software that manages interconnections between the user's software applications (Fig. 1). A standardized interaction mechanism abstracts away some details of the connection. Middleware operates in networked environments and provides interface *transparency* to the user. Thus, the designer needs to specify that Module-A is connected to Module-B, but they need not worry about the precise implementation, or even which processors the modules run on. Middleware may be integrated into the operating system, or it may exist as a separate "service" layer.

Designers of middleware systems usually select a particular interaction model as the basis for their system. Some examples include: distributed objects (e.g. CORBA,[6] DCOM[7]), message passing (e.g. MPI[8]), transactions (e.g. Tuxedo[9]), and distributed shared memory (e.g. BRAZOS[10]).

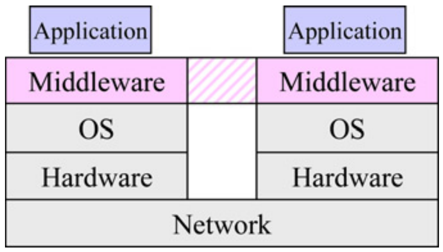

Fig. 1 The middleware concept. Middleware software allows user applications (modules A and B) to transparently span processor boundaries.

Distributed implementations of embedded software are on the rise. Researchers see opportunities to apply the techniques of conventional distributed computing to advance the capabilities of embedded physical devices.[11] Reconfigurable and composable systems are desirable goals,[12] but experts acknowledge that transitions are not always easy. Temporal performance analysis is generally not composable[13] and local optimization can easily lead to mediocre global behaviour.[14] In this section we examine several current approaches to real-time middleware.

1. Common Object Request Broker Architecture (CORBA)

CORBA is an open, distributed computing standard developed by the Object Management Group. As part of the general specification, it includes specific provision for real-time implementations.[6] CORBA is an object-based middleware service, designed to allow user client and server software to operate transparently with different, vendor-supplied transport mechanisms (i.e. the object request brokers or ORBs). So long as ORB implementations conform to the basic specifications, vendors are free to make different design and domain optimizations. Space does not permit an exhaustive discussion of CORBA architecture, but many tutorials are available.[‡]

CORBA is probably the single most popular middleware system in use today, thanks to its extensive library of optional services and cross-platform support.[11] A good survey of Real-Time CORBA research is provided by Fay-Wolfe, *et al*.[15] Several ORB implementations have been marketed as real-time, but many are still unsuited for hard real-time applications. Researchers at the University of Washington[16] have designed *The ACE ORB (TAO)* from scratch to provide predictable and differentiated services in hard real-time environments. Testing results from this show promise of achieving acceptable real-time performance.

2. Simplex

The Simplex architecture was developed at the Software Engineering Institute (SEI) at Carnegie Mellon University.[17] Simplex is a robust approach to reconfigurable control systems. The architecture allows designers to build modular control systems with variable levels of performance and analytic redundancy. Key components include I/O modules, control modules, switching logic, and the underlying communications system. Users provide several control modules, typically including some with low performance but high robustness. The switching logic is designed to exchange control-laws 'on-the-fly' if the safety or stability of the system is at risk. This also allows the user to upgrade these control modules, while the system is running with guarantees of system stability. Researchers have integrated simplex components with CORBA to create tele-laboratory environments.[18]

Supporting Simplex communications is a real-time publish-subscribe architecture.[19] It provides dependable real-time event-driven services between a network of processing nodes. Clients can publish or subscribe to named services. The designers provide enough transparency in the system operation to support conventional real-time analysis techniques.

3. GFLOPS and Middleware

Our middleware system is called the *GFLOPS Rapid Real-time Development Environment (GRRDE)*. The conceptual model behind GRRDE is a *publish-subscribe* architecture. Module interfaces are specified in terms of particular

[‡] *www.corba.org* is a good place to start

data-products. An output data-product is said to be *published*. Modules needing these data as inputs, can request a subscription from the GRRDE middleware. The data in question are then automatically delivered to the subscriber. Subscribers can request updates at preset intervals or whenever the source value changes. These services provide sufficient abstraction to reduce the effort needed to manage module connections, yet are concrete enough to maintain consistent mental models of operation. The publish-subscribe model also provides a more natural mapping than objects for many common flight software functions. During development it was felt that this particular approach to middleware services offered a better match to flight software applications than pre-existing alternatives.

Despite their popularity, CORBA and other object-based middleware may not be the best solution for all embedded applications. First, from an engineering perspective the overuse of object-oriented (OO) techniques has been criticized in high reliability systems.[20] Furthermore, the functional metaphor of OO-design is usually more appropriate to conventional data-oriented software than function oriented real-time software.[21] Second, testing and validation can be difficult. As systems become increasingly abstract the fine-grain testing required for flight systems becomes difficult. This problem is exacerbated if the ORB source-code is not available. Third, CORBA adds significant overhead to systems that may not need all of its properties. GRRDE services are designed to produce low computational and memory overhead.

Simplex and its underlying publish-subscribe middleware is the closest system to GRRDE in intended application and capabilities. Both seek to support embedded software development and both provide deterministic time guarantees. Three factors set GRRDE apart. First, GRRDE supports periodic subscriptions in addition to the more conventional aperiodic broadcast. Thus, GRRDE has provides greater service flexibility. Second, the ability to parameterize dispatch functions allows greater flexibility in message delivery. Third, our algorithms have been formally verified, using automata techniques, to ensure correctness and guarantee temporal properties.

C. General Timed Automata

General Timed Automata (GTA) is an analysis technique suitable for partially-synchronous, distributed systems. In this section we apply GTA modelling technique to the publish-subscribe services provided by GRRDE. This serves two objectives. First, by constructing an *abstract automaton* describing the GRRDE services, we have and unambiguous, formal specification of the properties that the system guarantees. Second, we produce hierarchical proofs that validate the *correctness* of the algorithms implemented in the GRRDE source code. We have proved basic timing and safety properties of both time-triggered and change-triggered contracts.

Hierarchical proofs can be used as a means of increasing the utility of automaton-based formal methods. Readers unfamiliar with the automaton-based proof techniques are encouraged to read Lynch.[4] This proof technique requires several steps. First, we specify an abstract, or *specification*, automaton. This automaton must satisfy the properties that we are interested in, yet make as little commitment as possible to the way in which the properties are satisfied. These properties are typically stated as logical invariants and proven through induction. Next, we formulate the detailed, or *implementation*, automaton. This may be a single automaton or a composition. Showing a rigorous correspondence, or *simulation relation*, between the states and behaviours of the specification and implementation automata, is enough to prove that the implementation automaton satisfies the same invariants and properties. The premise behind this approach is that it is usually more tractable to prove properties of the specification than the implementation itself.

This process can be repeated several times through successive steps of simulation and composition (Fig. 2). If a precise simulation relation can be provided for each stage, the underlying constructions can be formally verified. Thus, you could prove that your algorithm matches your specification, that your source code correctly implements an algorithm, and that your compiler produces exactly the correct machine instructions from code that you write.[22] In theory, this process can be extended all the way from specification to hardware, but the effort required to do this for realistic systems is prohibitive.

Several factors contribute to these limitations: state explosion makes large models unwieldy and the lack of formal semantics for some computer languages hampers the usefulness of the process. More importantly, the more complex the system, the easier it is to make mistakes in the formal analysis.[23] Finally, since most software-related failures can be traced to design and implementation errors,[2] conventional testing is less costly and more effective than exhaustive formal analysis during these development stages.

This leads us to consider how to effectively use formal analysis to support verification of the GRRDE services. How much analysis do we need? We have used GTA models to prove properties of the GRRDE service specifications

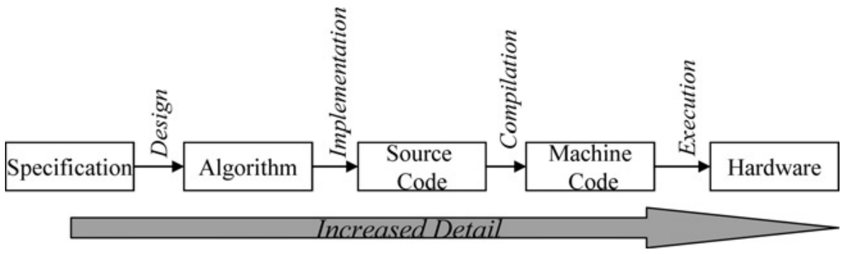

Fig. 2 Levels of abstraction in simulation proofs. If the semantics at each stage can be specified precisely, each level can be formally verified.

and algorithms. These stages are easiest to model and are most crucial to the elimination of system level errors. Thus, we gain the maximum benefit from a modest amount of effort.

II. Real-Time Services

GRRDE provides publish-subscribe communications services to user supplied software modules. This strategy affords the user great flexibility in sub-function design and implementation while reducing the effort spent in managing software interfaces. Two specific types of subscription services were implemented to support the development of the GFLOPS testbed: Time-Triggered subscriptions and Change-Triggered subscriptions. Each service type has its own semantics and its own range of applications.

A. Time-Triggered Subscriptions

Time-triggered contracts are most commonly encountered in low-level control systems. Situations where information updates are expected at regular intervals suggest the use of this type of contract. Specification of the service will include the update rate of the publishing module. Subscribers typically request a time-triggered contract at this rate or slower. Faster rate contracts are not prohibited, but are unlikely to be useful since they lead to duplicate values being delivered to the clients. Users must also be aware of the potential for under-sampling time domain phenomena when slow sampling rates are used.

Possible uses for time triggered contracts include:

- Sensor Polling
- Control Actuator Commanding
- Telemetry Logging
- Watchdog Services

B. Change-Triggered Subscriptions

Change triggered contracts are useful for data abstraction and status indication. For example, a sequencer may give the attitude control system (ACS) a command to change orientation. The *ACS status* variable may then indicate values of *Stewing* and then *CoursePointing* and finally *FinePointing* as the system stabilizes. Subscribers receive an updated copy of the subscribed variable whenever the value is changed. In essence, this type of service is a primitive form of multicast communication. These subscriptions are useful for communicating with supervisory modules as well as peers.

Possible uses include:

- Qualitative status or health updates
- Operating mode transitions
- Command feedback
- Multicast

III. Formal Service Specifications

Before we can prove any performance properties of the service specifications, we must first formally define how these services will work. Since this process can be quite lengthily we provide only a sketch of the full analysis. Interested readers may refer to[24] for a comprehensive treatment of the invariant proofs. We illustrate the analysis technique using the Time-Triggered specification and present a synopsis of the Change-Triggered analysis for comparison.

A. Formal Specification of Time-Triggered Services

A pictorial representation of the automata that comprise the time-triggered services shown in Fig. 3. These figures shows three types of automata. The finite set of user processes, U_i, $i \in I$, represent the interface to the publish-subscribe system (i represents a process index from the finite range I). We do not model any processing by the user, simply the interaction with the GRRDE service. Rather than depict the subscription selection, the service automaton, *Publish*, simply provides subscriptions to the single variable, X. We can view a full system as a parallel composition of several of these service automata, but a single type of subscription is sufficient for purposes of validation. The modelling of the delivery mechanism includes a reliable, *first-in-first-out (FIFO)* channel automaton C_i to capture effects of finite propagation time.

Three types of interactions are depicted. The first concerns the subscription process. This includes the *Subscribe*, *SubOk*, *Cancel*, and *CancelOk* actions. The time-triggered subscribe action is parameterized by the quantity, τ, which represents the requested delivery period. The *Cancel* action ends a current subscription and the other two actions are simply confirmations. The second type of interactions allows writing a new value, v, to X, and the associated confirmation. The last phenomenon modelled is the delivery process. Publish notifications are emitted from the publish server and propagate through the channel automaton C_i to the user. It should be noted that all of these actions are available to each user process; the figure just separates them for clarity.

1. *IOA Specification*

IOA pseudocode for the specification automaton is given in Fig. A1. This automaton encapsulates and publishes the shared variable $x \in X$. This variable is a tuple $X \equiv [v, seqno]$, $v \in V$, $seqno \in N$ where v is the actual data component of the service and is of arbitrary type, while *seqno* is a unique sequence number. Each action, *Write*(i, v'), causes a change in stored value of x such that $x'.v = v'$ and $x'.seqno = x.seqno + 1$. Sequence numbers allow us to precisely relate the contents of publish notifications to a unique write event.

The automata specifications presented in this section were developed with the *Input-Output Automata Toolkit (IOA Toolkit)*.[25] IOA defines a structured pseudocode in which the user can write automata models. This package performs syntax checking, type-setting and can interface with a number of theorem proving tools.

This automaton is a manual composition of the three types of automata. This composition includes the actual service automaton *Publish*, as well as the channel automata C_i, and the user automata, U_i. Since input actions are

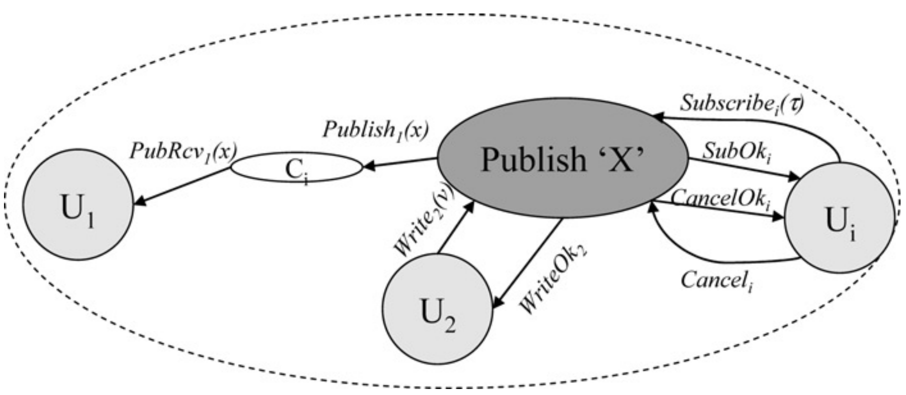

Fig. 3 Time-triggered publish composition. Model consists of user automata (U_i), as well as the service (Publish) and channel (C) automata.

subsumed by composition, all of actions are considered outputs.[§] The state declarations describe the internal state of the composed automaton. We compose the automaton in this way to permit analysis of the traces of the central *Publish* automaton. In this analysis the user automata exist only to maintain an external log of subscription notifications and to enforce well-formedness on system invocations.

A guide to interpreting the GTA pseudocode conventions can be found in Appendix A.

2. *Service Properties*

In order to prove useful properties of the time-triggered service, we follow a structured approach. We first establish a number of invariant properties of the composed automaton. These invariants allow us to prove theorems about the automata that capture the high-level properties of interest. These properties are summarized below. Sample proofs have been provided for the first few invariants in Appendix B. Similar inductive logic can be applied to the other properties.

Invariant 1: *In every reachable state of publish, all written values of x, save the initial value can be associated with a writing process, i.*

Invariant 2: *In every reachable state of automaton publish, all values, x' in transit to clients must have been previously written (i.e. appear in the WriteLog).*

Invariant 3: *In every reachable state of publish, all published values received by user automata must have been previously written (i.e. appear in the WriteLog).*

Invariant 4: *In every reachable state of Publish, the sequence numbers of messages in transit are non-decreasing.*

Invariant 5: *In every reachable state of Publish, the sequence numbers of messages delivered to user processes are non-decreasing.*

Together, these invariants are used to prove the first property about the time-triggered specification:

Theorem 1. *In every reachable state of Publish, the sequence of unique values recorded by subscribed users, constitute an ordered subset previous values of the shared variable x. I.e. $\forall i \in I$, $Unique(DestSeq_i.X) \subseteq \{WriteLog.X\}$.*

This theorem establishes temporal consistency of the values delivered to the clients. We require that the unique values delivered to the clients represent an ordered subset of the values written to the *Publish* automaton. This property is depicted graphically in Fig. 4. Thus, client records may skip or duplicate values written to the central automaton, but the sequence of values may not be reordered.

The second set of proofs is used to bound the temporal accuracy of the client values. This requires the following additional invariant:

Invariant 6: *In every reachable state of Publish, if a published value, x_1, is received at time, t_1, by a user process, then either $x_1 = PubValue$, or some process has written a new value to X with the last xmit.ub time units.*

Theorem 2. *In every reachable state of Publish, the each value delivered to the user automata reflects a true value of the publish variable within the preceding time window of width xmit.ub.*

This bound defines a sampling window. Sometime within this window, the true value of x must have been equal to the observed client value. This property is illustrated in Fig. 5.

The last set of invariants describe the periodic performance as of the subscription service.

Invariant 7: *In every reachable state of Publish, the current time is at least as great as the subscription-start time of any subscribed automaton.*

Invariant 8: *In every reachable state of Publish, the variable $PubCount_i$ accurately reflects the number of elements in the $DestSeq_i$ log.*

Invariant 9: *In every reachable state of Publish, only subscribed automata have non-empty message channels.*

Invariant 10: *In every reachable state of Publish, for all subscribed processes, the k_i-th publish event must occur between $StartTime_i \cdot k_i$ and $StartTime_i \cdot k_i + PubJitter$.*

[§] Readers unfamiliar with automata techniques are encouraged to refer to[4] for a summary of key principles and techniques.

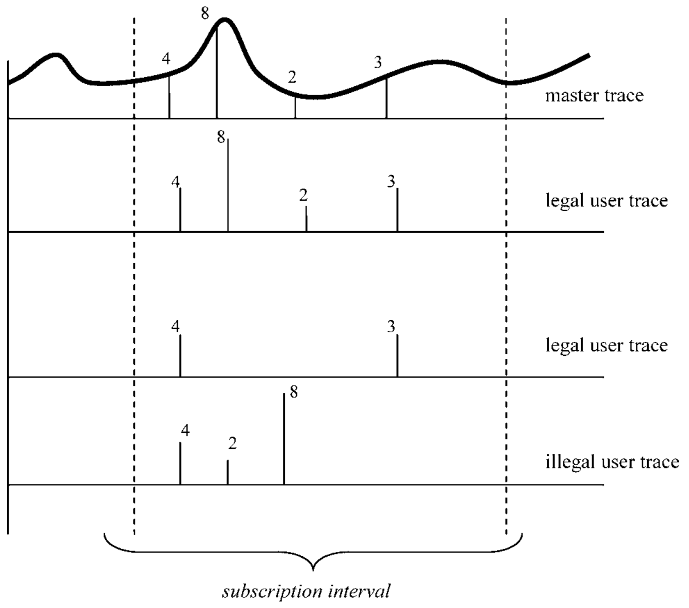

Fig. 4 Consistency of user traces. The master trace represents the values written to the shared variable *x*. Legal user traces must be consistent with the ordering of the master trace.

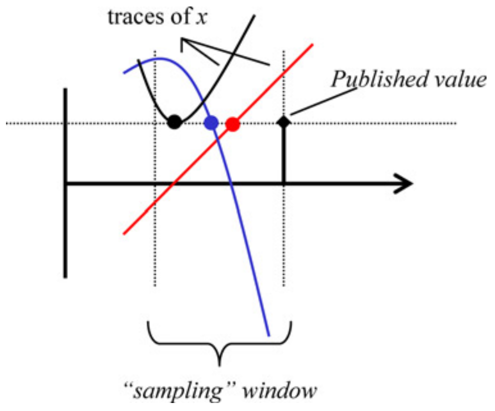

Fig. 5 Temporal accuracy of client values. At some point within the sampling window, the central value of *x* must have been equal to the value delivered to the client.

Invariant 11: *In every reachable state of Publish, the k_i-th publish message delivered to a subscribed user, is received between $now = StartTime_i + k_i \cdot Period_i + Xmit.lb$ and $now = StartTime_i + k_i \cdot Period_i + Xmit.ub + pub\_jitter$.*

Theorem 3. *Subscribers to time-triggered contracts receive periodic publish messages with bounded jitter $(pub\_jitter + (Xmit.ub - Xmit.lb))$.*

Thus, we are assured that the arrival of updates that each client receives is essentially periodic, with bounded variability.

To summarize these properties, we are assured that:

1. The values received by each client are a temporally consistent subset of the published values;
2. The values received by each client have a predictable temporal accuracy; and,
3. The clients will receive their updates periodically.

Since these properties agree with our intuitive understanding of the features a periodic publish-subscribe service should provide, we can be confident that our formal specification is well-constructed. We must note that this is only a specification (i.e. an unambiguous description of the service), not a means of implementing the service. We can, however, be confident that a system that accurately implements this specification, is assured of the same performance properties.

B. Formal Specification of Change-Triggered Services

The pictorial representation of the change-triggered service is shown in Fig. 6. The interface to the *PublishOnChange* automata is identical to the time-triggered version, except that the *Subscribe* event is not paramaterized. Clients receive publish events, not at regular intervals, but whenever the published value is changed. The formal GTA specification for the change-triggered service is given in Fig. A2. The logic involved in this specification is a little more complex than the time-triggered service, since each client must be guaranteed exactly one copy of each version of the published variable.

This change-triggered service shares many properties with the time-triggered service; Theorem 1 and Theorem 2, still hold. The unique feature of this specification is the 'exactly once' property mentioned above. The following invariants can be shown to hold:

Invariant 12: *In every reachable state of PublishOnChange, for each write action, and for each subscribed user, there will be a dispatched publish message. This message must be either: waiting to be dispatched, in transit, or received at the user.*

Invariant 13: *In every reachable state of, PublishOnChange, the dispatches spend a bounded amount of time in $DispatchSeq_i$ and $DestQueue_i$ and must reach the client within $DispatchBound.ub + Xmit.ub$ time units.*

Using these, together with the previous properties allows us to prove:

Theorem 4. *Any write invocation of the PublishOnChange automaton at time t_0, results in delivery of exactly one publish message to each subscribed client within the time window $t_0 + DispatchBound.lb + Xmit.lb \leq now \leq t_0 + DispatchBound.ub + Xmit.ub$.*

Thus, each subscribed client will receive one copy of each written value, in the same order the values were written, within a bounded time. Like the time-triggered service, these properties agree with our informal description of the change-triggered service. The formal analysis has allowed use to produce an unambiguous specification of the middleware services, and then prove that that specification will guarantee the properties of interest.

C. Formal Analysis of Service Algorithms

The automata proofs presented above establish properties of the service specification; any system that implements the specification, possesses those properties. As part of the development of the GRRDE system, we extended the

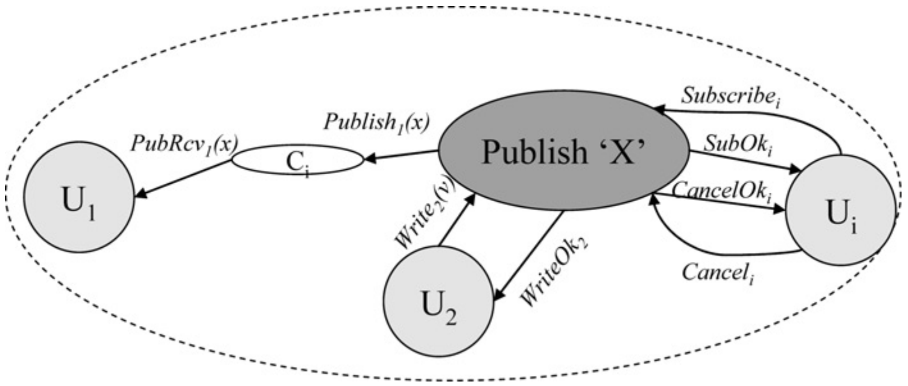

Fig. 6 The change-triggered automata. Identical to the time-triggered service, save that subscribe is not parameterized.

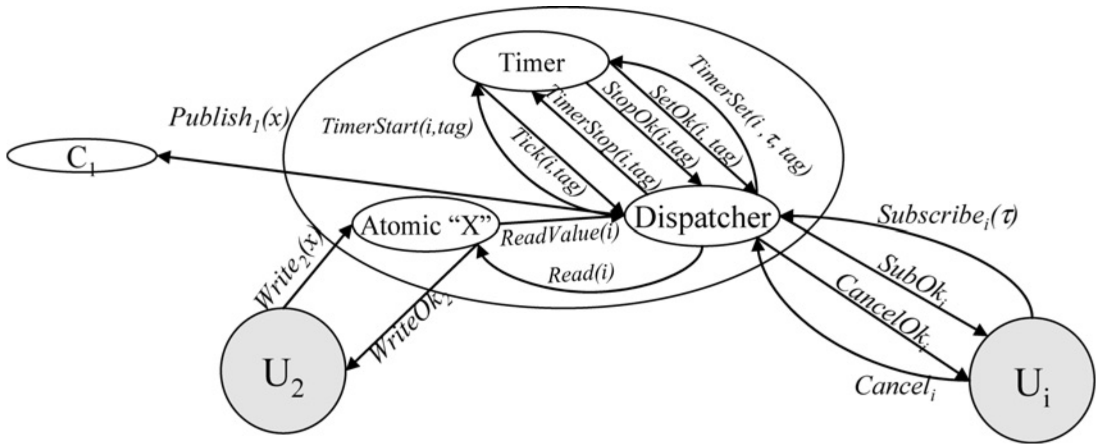

Fig. 7 Composition of the *publish* automaton.

formal analysis through another design stage—we formally verified the algorithm used to implement the specification. During this process the specification automaton (e.g. Fig. 3) is elaborated. As components, we use services provided by the operating system, the development language and other software tools. This allows us to create an automaton, composed of simpler objects, that together implement the service specifications (Fig. 7). This automaton is essentially a representation of the algorithm used to implement the service.

The approach to validating the implementation algorithm against the specification algorithm uses a technique known as *simulation*. Simulation, in the context of GTA theory, is a means of establishing correspondence between an abstract specification and its implementation. Specifically, we must shown two things:

- That the external actions of the two automata are identical, and
- At any point in time, the states of the specification can be uniquely derived from the states of the implementation.

Space does not permit a full presentation of the simulation proofs. Instead, we shall summarize the key relations that were established during this process. First, the simulation analysis proved that the algorithms designed for GRRDE would correctly implement the specification of the middleware services. Second, composing the implementation automata from known primitives (e.g. operating services) gave a clearer meaning to the abstract time-bounds mentioned in the specification.

- The publication jitter for the time-triggered *publish* automata will vary linearly with the number of subscribers:

$$PubJitter = x_1 \cdot N + x_0 \tag{1}$$

  Thus the variability in arrival times increases with the number of client processes. Testing on a network of 400MHz embedded processors yielded evidence of this linear relation and permitted us to measure the scaling constants: $x_0 = 1 \times 10^{-6}$ s and $x_1 = 1 \times 10^{-7}$ s.
- The upper-bound on the delivery delay Δ_{total}, also varies linearly with the number of subscribed clients:

$$\Delta_{total} = y_0 + y_1 \cdot N \tag{2}$$

  Similar testing gave: $y_0 = 275 \times 10^{-6}$ s and $y_1 = 11 \times 10^{-6}$ s. These bounds are quite large, but further investigation suggested that access to specialized operating system services accounted for much of the delay. An optimized implementation could reduce this time.

IV. Conclusions

Safety-critical, real-time systems such as spacecraft rely on strong guarantees of determinism to ensure correct operation. Operating within this context, and aware of the need for high dependability, the GRRDE middleware services were designed and implemented to reduce an engineer's non-productive workload. GRRDE's abstract publish-subscribe services come in two varieties. Periodic subscriptions deliver data at regular intervals and can operate independently from the publisher. This type of software connection is frequently found in control systems. Aperiodic subscriptions function like a multi-cast group communication and are well suited for reporting module status or the like.

Embedded applications demand more from software than just working demonstrations. The GRRDE services were validated with formal methods—demonstrating the usefulness of formal approaches in flight software engineering. Other investigations used GRRDE to develop simulations of distributed spacecraft systems. These companion studies highlighted the usefulness of aerospace middleware.

A. Companion Studies

Our application studies provide concrete illustrations of GRRDE's potential to reduce software interface complexity. Our examples are apt to be the first step in convincing a program manager to consider using GRRDE in a mission setting. This demonstrations answered the question: "What can this technology do for my mission?"

1. Simulations. This study examined the application of GRRDE to a large, complex simulation. The middleware managed interconnections between many software modules, developed by different people, serving very different functions.

   The mission chosen for this study was the TechSat21 technology demonstrator mission. TechSat21 was a distributed aperture radar concept. Our simulation scenario began with the precision formation flight of a small cluster of satellites. As the target area of Earth comes into view, the satellites activated their radar transmitters, coordinated reception, exchanged the returned signals and synthesized the ground scene.
2. Autonomy. This study expanded the role of GRRDE as a technology enabler. We demonstrated the flexibility of the GRRDE middleware to integrate an autonomous fault diagnosis engine into a simple simulation. The flexible communications abstractions allow the new functions of the diagnostic engine to be layered on top of conventional flight software. GRRDE assists this integration in two ways. First, the abstract services allow us to tap directly into the telemetry stream. Second, real-time data delivery guarantees, reduce the potential for state confusion due to temporal inconsistency.
3. Automatic Code Generation. One common innovation in the development of embedded control systems is the use of automatic code generation. These techniques and tools allow direct generation of real-time software from block-diagram models of monitor and control systems. This study examined the integration of GRRDE middleware with a common code generation package. We use the Real-Time Workshop (RTW) and Target Language Compiler (TLC) components of Mathworks's Simulink software to directly generate GRRDE-compatible software modules from their Simulink representations. This method was extremely effective and can be used to rapidly build libraries of interoperable flight software modules or simulations.

B. Final Comments

Conventional flight software methods may be effective for near-term missions, but the tendency to rely on software to implement increasingly complex functions suggests limits to these techniques. As system complexity grows, the costs of software development and the likelihood of software failures will increase as well. Unless measures are taken to manage complexity and promote reliability, extensibility and scalability, flight software may limit mission capabilities rather than enable them. Middleware systems have been employed to good effect in terrestrial settings, and are now being adopted into embedded, real-time applications. Approaches such as those used in developing GRRDE, can be used to bring some terrestrial innovations into the spacecraft forum.

Appendix A.
A Guide to GTA Pseudocode

There are several types of state variables in the pseudocode:

- Constants. Although some quantities can be specified by the designer, constants often represent a physical property of the system (e.g a response time). These variable are used mostly to specify parameters in invariant statements. Numeric values can be determined through testing or further analysis.
- Time Bounds. GTA automata use explicit variables such as *first* or *last* to restrict when certain actions can take place. These variables are a standard feature of GTA models, but do not represent "real" variables.

```
%Constants
SubOk :  TimeBound
CancelOk : TimeBound
WriteOk : TimeBound
Xmit : TimeBound
WriteCommitTime : Timebound
PubJitter: Real
automaton Publish($V_0$ : V)
    signature
        output
            Subscribe$_i$($\tau$: Real)
            Publish$_i$(T: TxRec)
            PubRcv$_i$(T: TxRec)
            Cancel$_i$
            SubOk$_i$
            CancelOk$_i$
            Write$_i$($v$: V)
            WriteOk$_i$
        internal
            WriteCommit$_i$
        time-passage
            $v$ ($t$: Real)
    states
        %Records at each client node
        DestSeq[$i$:I]: an array of RcvRec sequences, each initially empty,
        PubCount[$i$:I]: an array of integers, each initially 0
        StartTime[$i$:I]: an array of reals, each initially 0

        %Transit Queues
        DestQueue[$i$:I]: an array of TxRec sequences, each initially empty

        %Housekeeping to ensure well-formedness
        Subscribed[$i$:I]: an array of booleans, each initially false
        PendingCommand[$i$:I]: an array of commands, each initially nil
        Period[$i$:I]: an array of reals, each initially 0
        PubStarted[$i$:I]: an array of booleans, each initially false
        %Records at server
        WriteLog: a sequence of WriteRecs, initially {[[$V_0$,0],null,0]}

        %Published Variable
        PubValue: a DataRec, initially [$V_0$,0]
        PendingWriteValue[$i$:I]: an array of V

        %Time and GTA bounds
        now: a non-negative real, initially 0,
        FirstCancelOk[$i$:I]: an array of reals, each initially 0
        LastCancelOk[$i$:I]: an array of reals, each initially $\infty$
        FirstPublish[$i$:I]: an array of reals, each initially 0
        LastPublish[$i$:I]: an array of reals, each initially $\infty$
        FirstSubOk[$i$:I]: an array of reals, each initially 0
        LastSubOk[$i$:I]: an array of reals, each initially $\infty$
        FirstWriteOk[$i$:I]: an array of reals, each initially 0
        LastWriteOk[$i$:I]: an array of reals, each initially $\infty$
        FirstWriteCommit[$i$:I]: an array of reals, each initially 0
        LastWriteCommit[$i$:I]:an array of reals, each initially $\infty$
    transitions
        output Subscribe$_i$($\tau$:Real)
           pre PendingCommand[i] = nil $\wedge$ $\neg$ Subscribed[i]
           eff PendingCommand[i] := Sub
           Subscibed[i] := true
           Period[i] := $\tau$
           StartTime[i] := now
           FirstSubOk[i] := now + SubOk.lb
           LastSubOk[i] := now + SubOk.ub
           FirstPublish[i] := now + Period[i]
           LastPublish[i] := FirstPublish[i] + PubJitter
```

Fig. A1 The Publish automaton specification.

```
output SubOk$_i$
   pre PendingCommand[i] = Sub $\wedge$ now $\geq$ FirstSubOk[i]
   eff PendingCommand[i] := nil
   FirstSubOk[i] := 0
   LastSubOk[i] := $\infty$
output Publish$_i$(T:TxRec)
   pre Subscribed[i]    $\wedge$ now $\geq$ FirstPublish[i]
                        $\wedge$ T = [PubValue,[now + Xmit.lb,now + Xmit.ub]]
   eff DestQueue[i] := DestQueue[i] $\vdash$ T
   FirstPublish[i] : = FirstPublish[i] + Period[i]
   LastPublish[i] := FirstPublish[i] + PubJitter

output PubRcv$_i$(T: TxRec)
   pre T = head(DestQueue[i])  $\wedge$ now $\geq$ head(DestQueue[i]).delivery.lb
                               $\wedge$  PendingCommand[i] $\neq$ Sub
   eff DestQueue[i] := tail(DestQueue[i])
   PubCount[i] := PubCount[i] + 1
   DestSeq[i] := DestSeq[i] $\vdash$ [T.X, PubCount[i], now]
output Cancel$_i$
   pre Subscribed[i]
   eff Subscribed[i] := false
   PendingCommand[i] := Cancel
   FirstPublish[i] := 0
   LastPublish[i] := $\infty$
   FirstCancelOk[i] = now + CancelOk.lb
   LastCancelOk[i] = now + CancelOk.ub

output CancelOk$_i$
   pre PendingCommand[i] = Cancel $\wedge$ now $\geq$ FirstCancelOk[i]
                                   $\wedge$  DestQueue[i] = {}
   eff PendingCommand[i] := nil
   PubCount[i] := 0
   DestSeq[i] := {}
   FirstCancelOk[i] := 0;
   LastCancelOk[i] := $\infty$
output Write$_i$($v$:V)
   pre PendingCommand[i] = nil
   eff PendingCommand[i] = Commit
   PendingWriteValue[i] := $v$
   FirstWriteCommit[i] := now + WriteCommitTime.lb
   LastWriteCommit[i] := now + WriteCommitTime.ub
internal WriteCommit$_i$
   pre PendingCommand[i] = Commit
   eff PubValue := [PendingWriteValue[i],PubValue.SeqNo + 1]
   WriteLog[i] := WriteLog $\vdash$ [pubValue,i,now]
   PendingCommand[i] = Write;
   FirstWriteOk[i] = now + WriteOk.lb
   LastWriteOk[i] = now + WriteOk.ub
   FirstWriteCommit[i] := 0
   LastWriteCommit[i] := $\infty$
output WriteOk$_i$
   pre PendingCommand[i] = Write $\wedge$ now $\geq$ FirstWriteOk[i]
   eff PendingCommand[i] := nil;
   FirstWriteOk[i] := 0;
   LastWriteOk[i] := $\infty$
time-passage $v$ ($t$:Real)
   pre ($\forall$ i:I ((now + t) < LastWriteOk[i]
         (now + t) < LastCancelOk[i]
         (now + t) < LastSubOk[i]
         (now + t) < LastPublishOk[i]
         (now + t) < LastWriteCommit[i]
         length(DestQueue[i]) > 0 $\Rightarrow$
              ((now + t) < head(DestQueue[i]).Delivery.ub))))
   eff now := now + t
```

Fig. A1 Continued.

```
%Constants
SubOk :  TimeBound
CancelOk : TimeBound
WriteOk : TimeBound
Xmit : TimeBound
WriteCommitTime : TimeBound
PubJitter: Real
DispatchBound : TimeBound
GenerateBound: TimeBound
automaton PublishOnChange($V_0$ : V)
    signature
        output
            Subscribe$_i$
            SubOk$_i$
            Publish$_i$ (T: TxRec)
            PubRcv$_i$ (T: TxRec)
            Cancel$_i$
            CancelOk$_i$
            Write$_i$ ($v$ : V)
            WriteOk$_i$
        internal
            WriteCommit$_i$
            GenerateMessages$_i$
        time-passage
            $v$ ($t$ : Real)

    states
        %Records at each client node
        DestSeq[$i$:I]: an array of RcvRec sequences, each initially empty
        PubCount[$i$:I]: an array of integers, each initially 0
        StartTime[$i$:I]: an array of reals, each initially 0

        %Transit Queues
        DestQueue[$i$:I]: an array of TxRec sequences, each initially empty
        DispatchSeq[$i$:I]: an array of TxRec sequences, each initially empty

        %Housekeeping to ensure well-formedness
        Subscribed[$i$:I]: an array of booleans, each initially false
        PendingCommand[$i$:I]: an array of commands, each initially nil
        PubStarted[$i$:I]: an array of booleans, each initially false
        Generating: an client index I or nil, initially nil

        %Records at server
        WriteLog: a sequence of WriteRecs, initially {[[$V_0$,0],null,0]}

        %Published Variable
        PubValue: a DataRec, initially [$V_0$,0]
        PendingWriteValue[$i$:I]: an array of V

        %Time and GTA bounds
        now: a non-negative real, initially 0,
        FirstCancelOk[$i$:I]: an array of reals, each initially 0
        LastCancelOk[$i$:I]: an array of reals, each initially $\infty$
        FirstSubOk[$i$:I]: an array of reals, each initially 0
        LastSubOk[$i$:I]: an array of reals, each initially $\infty$
        FirstWriteOk[$i$:I]: an array of reals, each initially 0
        LastWriteOk[$i$:I]: an array of reals, each initially $\infty$
        FirstWriteCommit[$i$:I]: an array of reals, each initially 0
        LastWriteCommit[$i$:I]:an array of reals, each initially $\infty$
        FirstGenerate[$i$:I]: an array of reals, each initially 0
        LastGenerate[$i$:I]:an array of reals, each initially $\infty$
```

Fig. A2 The *PublishOnChange* automaton specification.

```
transitions
   output Subscribe$_i$
      pre PendingCommand[i] = nil $\wedge$ $\neg$ Subscribed[i]
      eff PendingCommand[i] := Sub
      Subscibed[i] := true
      StartTime[i] := now
      FirstSubOk[i] := now + SubOk.lb
      LastSubOk[i] := now + SubOk.ub
   output SubOk$_i$
      pre PendingCommand[i] = Sub $\wedge$ now $\geq$ FirstSubOk[i]
      eff PendingCommand[i] := nil
      FirstSubOk[i] := 0
      LastSubOk[i] := $\infty$
   output Publish$_i$(T:TxRec)
      pre length(DispatchSeq[i] $\neq$ 0
               $\wedge$ now $\geq$ head(DispatchSeq[i]).Deliery.lb
               $\wedge$ T = [head(DispatchSeq[i]).X,[now + Xmit.lb,now + Xmit.ub]]
      eff DestQueue[i] := DestQueue[i] $\vdash$ T
      DispatchSeq[i] := tail(DispatchSeq[i])
   output PubRcv$_i$(T: TxRec)
      pre T = head(DestQueue[i]) $\neq$ 0
               $\wedge$ now $\geq$ head(DestQueue[i]).delivery.lb
               $\wedge$ PendingCommand[i] $\neq$ Sub
      eff DestQueue[i] := tail(DestQueue[i])
      PubCount[i] := PubCount[i] + 1
      DestSeq[i] := DestSeq[i] $\vdash$ [T.X, PubCount[i], now]

   output Cancel
      pre Subscribed[i] $\wedge$ PendingCommand[i] = nil
      eff Subscribed[i] := false
      PendingCommand[i] := Cancel
      FirstCancelOk[i] = now + CancelOk.lb
      LastCancelOk[i] = now + CancelOk.ub
   output CancelOk
      pre PendingCommand[i] = Cancel $\wedge$ now $\geq$ FirstCancelOk[i]
                                  $\wedge$ DestQueue[i] = {}
                                  $\wedge$ DispatchSeq[i] = {}
      eff PendingCommand[i] := nil
      PubCount[i] := 0
      DestSeq[i] := {}
      FirstCancelOk[i] := 0;
      LastCancelOk[i] := $\infty$
   output Write$_i$($v$:V)
      pre PendingCommand[i] = nil
      eff PendingCommand[i] = Commit
      PendingWriteValue[i] := $v$
      FirstWriteCommit[i] := now + WriteCommitTime.lb
      LastWriteCommit[i] := now + WriteCommitTime.ub
   internal WriteCommit$_i$
      pre PendingCommand[i] = Commit $\wedge$ now $\geq$ FirstWriteCommit[i]
                                  $\wedge$ Generating = nil
      eff pub_value := [PendingWriteValue[i],PubValue.SeqNo + 1]
      WriteLog[i] := WriteLog $\vdash$ [PubValue,i,now]
      PendingCommand[i] := Write;
      Generating := i
      FirstGenerate[i] := now + GenerateBound.lb
      LastGenerate[i] := now + GenerateBound.ub
      FirstWriteOk[i] := $\infty$      %Inhibit write until generation
      LastWriteOk[i] := $\infty$
      FirstWriteCommit[i] := 0
      LastWriteCommit[i] := $\infty$
```

Fig. A2 Continued.

```
internal GenerateMessages$_i$
   pre Generating = i ∧ now ≥ FirstGenerate[i]
   eff for j:I so that Subscribed[j] do
       DispatchSeq[j] := DispatchSeq[j] ⊢
              [PubValue,[now + DispatchBound.lb,now + DispatchBound.ub]]
   od
   Generating := nil
   FirstGenerate[i] := 0
   LastGenerate[i] := ∞
   FirstWriteOk[i] := now + WriteOk.lb
   LastWriteOk[i] := now + WriteOK.ub

output WriteOk$_i$
   pre PendingCommand[i] = write ∧ now ≥ FirstWriteOk[i]
   eff PendingCommand[i] := nil;
   FirstWriteOk[i] := 0;
   LastWriteOk[i] := ∞

time-passage $v$ ($t$:Real)
   pre     (∀ i:I ((now + t) < LastWriteOk[i]
           (now + t) < LastCancelOk[i]
           (now + t) < LastSubOk[i]
           (now + t) < LastGenerate[i]
           (now + t) < LastWriteCommit[i]
           length(DestQueue[i]) > 0 ⇒
               ((now + t) < head(DestQueue[i]).Delivery.ub)
           length(DispatchSeq[i]) > 0 ⇒
               ((now + t) < head(DispatchSeq[i]).Delivery.ub)))
   eff now := now + t
```

Fig. A2 Continued.

- Log Variables. Many interesting automaton properties are naturally described by discussing the system traces and execution (i.e. the steps an algorithm takes). However, state invariants are the most straightforward properties to prove. We can convert trace properties into state invariants by introducing artificial log variables. Thus, as an automaton takes an action, it records an entry in this state variable. This useful contrivance creates a common framework for all of our proofs, but these variables are not reflected in our software.
- Constraints. Well-formedness assumptions about the automaton interactions can be made explicit in the specification. Certain state variables are attributed to the user processes and inhibit further commands while confirmation for a current command is still pending.
- Core Variables. These state variables are intended to have some physical realization in an implemented system. Abstract, specification automata will typically have few state components of this type.

We also define the following additional variable types:

- *WriteRec* $\equiv [x, i, t]$, $x \in X$, $i \in I$, $t \in \Re^{\geq 0}$. This tuple is used to keep records of the incoming *write* actions. We record the value, x, the process index, i, and the time t.
- *TimeBound* $\equiv [lb, ub]$, $lb, ub, \in \Re^{\geq 0}$ This tuple is used to record lower and upper timing bounds. These variables can record relative or absolute time.
- *TxRec* $\equiv [x, bound]$, $x \in X$, $bound \in TimeBound$. These tuples are used to transmit published values through the transit queues from the service automaton to the clients. The first element contains the published data while the second gives the earliest and latest delivery times.
- *RcvRec* $\equiv [x, k, t]$, $x \in X$, $k \in Z^+$, $t \in \Re^{\geq 0}$. These variables are used to log message delivery in the user automata. We record the data, x, a message count, k, and the arrival time, t.

Appendix B.
Invariant Proof Example

The following examples illustrate the use of logical induction to prove properties about the states of the service automata. The basic approach is to first show that the property is initially true. Once this is established we assume the property holds in state s and show that after any action π, the invariant still holds. The tuple of the old-state, the action and the new-state is denoted (s, π, s'). The use of sequential log variables is a convenient artiface that lets us reason about the external traces of the automata.

Proof of Invariant 1: This is a fairly trivial assertion and is formally stated as Invariant I1. We consider an induction on the states of *Publish*. We can see from the automaton specification that the invariant holds in the start state. Assuming that the invariant holds in state s we consider the transition (s, π, s') and prove that the invariant hold in the final state s'. The property is vacuously true in all actions, π, save *write*, since no other actions alter the *WriteLog* variable. Considering this case in more detail we have:

- Case $\pi = \mathit{write}(i, v)$. The invariant is clearly true in the final state since the appended element of *WriteLog* is tagged with the process index i.

Therefore the invariant assertion holds for all reachable states of *Publish*.

Proof of Invariant 2: Consider induction on the states of *publish*. Since the message channels ($\mathit{DestQueue}_i$) are initially empty, the assertion is clearly true in the initial state. Considering the inductive step we examine the transition (s, π, s'). The assertion is vacuously true for all transitions save $\mathit{publish}_i(T)$. and $\mathit{pubrcv}_i(T)$. We examine these transitions separately.

- Case $\pi = \mathit{publish}_i(T)$. Since the transition specifies that $T \cdot x = \mathit{head}(\mathit{WriteLog}) \cdot x$, the final state s' must satisfy the invariant.
- Case $\pi = \mathit{pubrcv}_i(T)$. This action only removes elements from $\mathit{DestQueue}_i$, thus s' must satisfy I2.

Proof of Invariant 3: We consider induction on the states of *publish*. This invariant is I3. In the initial state of *publish*, the client records are empty so the assertion is clearly true. For the inductive step in the proof we consider the state transition (s, π, s'). The only actions that modify the client record $\mathit{DestSeq}_i$ are $\mathit{cancel\_ok}(i)$ and $\mathit{pub\_rcv}(i, T)$. For the other actions, the invariant assertion is vacuously true.

- Case $\pi = \mathit{cancel\_ok}_i$. Since in s', the client log is empty, i.e. $\neg\exists R$, $R \in \mathit{DestSeq}_i$, and the invariant I3 still holds.
- Case $\pi = \mathit{pub\_rcv}_i(T)$. We know from I2 that all entries in the channel queues must correspond to *WriteLog* entries. Therefore when these messages are received by the user processes, the correspondence must be maintained.

Acknowledgments

The authors would like to thank the sponsor of this work: The Air Force Office of Scientific Research for Generalized Flight Operations Processing Simulator—Contract F49620-99-1-0217, under the technical supervision of Dr. Robert Herklotz.

References

[1]Leveson, N., "Systemic Factors in Software-Related Spacecraft Accidents", *AIAA Space 2001 Conference and Exposition, Albuquerque*, NM, 28–30 Aug. 2001, AIAA 2001-4763.f

[2]Lutz, W., "Analyzing Software Requirements Errors in Safety-Critical Embedded Systems", *IEEE International Symposium on Requirements Engineering*, 1993, pp. 126–133.

[3]Mills, M.,"A Call from the Heavens Above", *Washington Post*, Nov. 23, 1998, p. F23.

[4]Lynch, N. A., *Distributed Algorithms*, Morgan Kaufman Publishers, Inc., CA, 1996. Chap. 23

[5]Enright, J., Sedwick, R., and Miller, D., "High Fidelity Simulation for Spacecraft Autonomy Development", *Canadian Aeronautics and Space Journal*, Vol.48, No. 1, March 2002.

[6]The Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Rev 2.4.1 November 2000, pp. 871–927

[7]Hortsmann, M., and Kirtland, M., "DCOM Architecture", Microsoft Corporation, MSDN Library, 1997

[8]Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. J., *MPI: The Complete Reference*, MIT Press, 1995

[9]Hall, C., *Building Client/Server Applications Using Tuxedo*, Wiley, 1996.

[10]Speight, E., and Bennett, J. K., "Using Multicast and Multithreading to Reduce Communication in Software DSM Systems", *IEEE High-Performance Computer Architecture*, 1998. Proceedings., pp. 312–322

[11]Bates, J., "The State of the Art in Distributed and Dependable Computing", *CaberNet Report*, Laboratory for Communications Engineering: University of Cambridge, 1998.

[12]Kopetz, H., "Software Engineering for Real-Time: A Roadmap", *Proceedings of the (ACM) Conference on the Future of Software Engineering*, 2000, Limerick, Ireland., pp. 201–211.

[13]Lee, E., "What's Ahead for Embedded Software?", *IEEE Computer Magazine*, September 2000, pp. 18–26.

[14]Sha, L., Rajkumar, R., and Sathaye, S., "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Distributed Systems", *Proceedings of the IEEE*, Vol. 82, No. 1, January 1994.

[15]Fay-Wolfe, V., DiPoppo, L., Cooper, G., Johnston, R., Kortmann, P., and Thuraisingham, B., "Real-Time CORBA", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 11, No. 10, October 2000, pp. 1073–1089.

[16]Schmidt, D., Levine, D., and Mungee, S., "The Design of the TAO Real-Time Object Request Broker," *Computer Comm. J.*, Summer 1997.

[17]Seto, D., Krogh, B., Sha, L., and Chutinan, A., "The Simplex Architecture for Safe On-Line Control System Upgrades", *Proceedings of the American Control Conference*, June 1998, pp. 3504–3508.

[18]Polze, A., Schwarz, J., Wehner, K., and Sha, L., "Integration of CORBA Services with a Dynamic Real-Time Architecture", 6th IEEE Real-Time Technology and Applications Symposium, 2000, pp. 198–206

[19]Rajkumar, R., Gagliardi, M., and Sha, L., "The Real-Time Publish/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation", First IEEE Real-Time Technology and Applications Symposium, May 1995, pp. 66–75.

[20]Hatton, L., "Does OO Synch with How We Think?", *IEEE Software*, May/June 1998.

[21]Wright, D. T., and Williams, D. J., "Object-like Software Design Methods for Intelligent Real-time Process Control," *Intelligent Control, 1993*., *Proceedings of the 1993 IEEE International Symposium on*, 1993, pp. 144–149.

[22]Erkkinen, T., "Safety-Critical Software Generation", *Proceedings of the IEEE International Symposium on Computer Aided Control Systems Design*, Kohala Coast, Hawaii, 1999, pp. 237–242.

[23]Tanenbaum, A., "In Defense of Program Testing or Correctness Proofs Considered Harmful", *SIGPLAN Notices*, Vol. 11 No 5.

[24]Enright, J., *A Flight Software Development and Simulation Framework for Advanced Space Systems*, MIT-Space Systems Laboratory Report SSL-#5-2002,

[25]Garland, S., and Lynch, N., "Using I/O Automata for Developing Distributed Systems", *Foundations of Component-Based Systems*, Gary T. Leavens and Murali Sitaraman, Cambridge University Press, 2000, pp. 285–312.

Michael Hinchey
Associate Editor