A Language for Unified Verification and Implementation for Distributed Avionics

Benjamin F. Jones[a] and Lee Pike[b]
Galois, Inc., Portland, Oregon, 97204

Srivatsan Varadarajan[c] and Brendan Hall[d]
Honeywell Advanced Technology, Golden Valley, Minnesota, 55422

In designing, verifying, and validating distributed systems today, an engineer is often faced with having to specify a system multiple times. For example, the engineer might specify it once in a model-checker for formal analysis, in an architectural description language for requirements analysis, and in a programming language for testing. By specifying the same system multiple times, there is risk that each specification has different semantics, so that the system tested differs from the one verified, for example. To help solve this problem, we envision an *architectural domain-specific language* (ADSL), or a unified language from which formal models, executable code, and architectural models can be synthesized. To make our problem tractable, we focus on distributed fault-tolerant systems. We present the *Language for Integrated Modeling and Analysis* (LIMA), a particular ADSL that we have designed. We describe LIMA and its application to case-studies motivated by avionics design.

[a] Senior Research Engineer, 421 SW 6th Ave.
[b] Research Lead
[c] Staff Scientist
[d] Engineer Fellow

I. Introduction

Modern avionic systems continue to grow larger in size and complexity, and state-of-the-art flight control systems can consist of more than a million lines of code. Lockheed Martin's F22 Raptor contains approximately 1.7M lines of code [1] and the next-generation F35 fighter is estimated to comprise 5.7M lines of code. This trend is not limited to the military arena; the software content of the Boeing 787 is estimated at 13 million lines of code [2].

Concurrently, advancements in networking technology enable increasingly distributed systems, and network-centric Integrated Modular Avionics (IMA) architectures are now the industry norm across all aircraft segments, from large air transport A380 [3] to general aviation [4]. This integration of multiple aircraft functions into IMA architectures offers many benefits. Leveraging common hardware may enable significant SWaP (size weight and power) reduction. Standardization on hardware platforms may support the improved optimization of the system obsolescence management. Finally, the ability to support more cooperation between traditionally federated aircraft functions may support greater efficiency and safety. The benefits of such integration are argued in [5].

Industrial architectures often evolve and are usually based on informal assumptions. By establishing a formal model of the system, we can uncover many undocumented assumptions. Once formal models are developed, we have found that the application of formal methods can systematically uncover edge cases and erroneous behavior that are not otherwise obvious. For example, in our previous work developing fault-tolerant protocols [6, 7], we found that the application of model checkers is particularly valuable for uncovering erroneous edge cases in the protocol logic. In [6], we further demonstrated the ability to leverage the formal models developed during protocol design to generate system-level test cases.

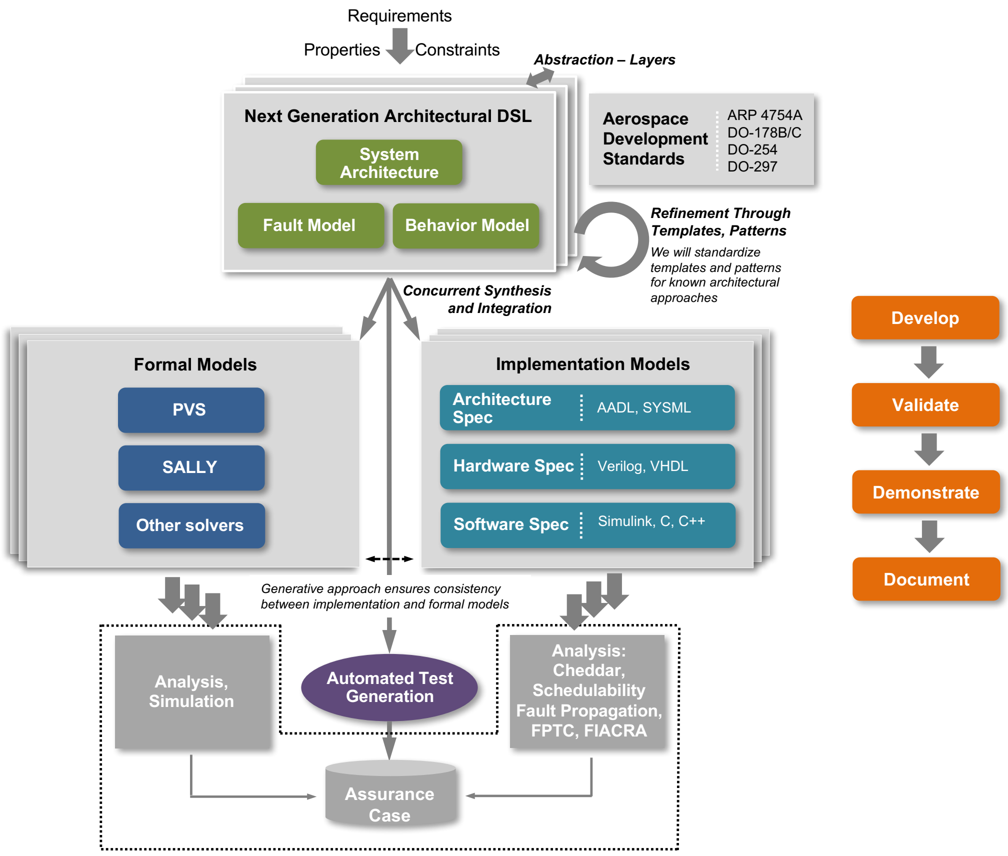
Formal modeling is typically—but not exclusively—used to reason about behavior. Architectural modeling helps to document and reason about non-functional properties. Numerous architectural languages, including AADL[8], EAST-ADL [9], SLIM [10], and SYSML [11] have emerged in the past decade. Analysis tools support the models developed using architectural modeling notations. These tools support many aspects of system examination from schedulability [12] analysis to failure and propagation analysis (HiHOPs [13] and ADAPT [14]), and automated fault-tree generation [15, 16].

While introducing formal or architectural modeling into a distributed system development effort can improve the documentation and quality of the system, it comes at a high cost. In particular, it means there are three separate artifacts to maintain and keep consistent: the formal model, the architectural model, and the actual system implementation. Ensuring their consistency is ad-hoc, as each language has its own syntax and semantics with no connections between them. At best, the additional effort required to keep the models consistent outweighs the benefits of increased assurance. At worst, it provides a false sense of assurance, where the implementation does not satisfy properties specified in architectural or formal models.

The central thesis of the research described here is that formal models and analysis is not cost effective unless those models are integrated into the software development process so that there is one central view of the system and the models and analysis are directly connected with the software. Without a formal link through the system refinement and implementation processes, we risk the *abstraction gap* where implementation details may impact the assumptions of the higher level abstract model of the system that has been formally argued. We call the specification that generates models and implementations an *architectural domain-specific language* (ADSL). The user needs only to specify the system in one language, and from that, has multiple "views" onto the system.

Moreover, while the work we describe here focuses on connecting executable software with formal models for verification, it is in service of our vision for an architectural workbench as shown in Figure 1. We envision an ADSL from which multiple analyses are available including

- Synthesizing formal models (e.g., PVS [17] for interactive theorem-proving or Sally [18] for model checking);
- Synthesizing architectural models (e.g., SysML [19], AADL [20, 21]), hardware models (e.g., Verilog or VHDL), and software models (e.g., Simulink or C/C++);
- Automate test generation for system testing from ADSL system models;
- Integrate into assurance case toolsets (e.g., [22, 23]) to systematically integrate the formal sys-



Requirements
Properties
Constraints
Abstraction – Layers
Next Generation Architectural DSL
System Architecture
Fault Model
Behavior Model
Aerospace Development Standards
ARP 4754A
DO-178B/C
DO-254
DO-297
Refinement Through Templates, Patterns
We will standardize templates and patterns for known architectural approaches
Concurrent Synthesis and Integration
Formal Models
PVS
SALLY
Other solvers
Implementation Models
Architecture Spec
AADL, SYSML
Hardware Spec
Verilog, VHDL
Software Spec
Simulink, C, C++
Develop
Validate
Demonstrate
Document
Generative approach ensures consistency between implementation and formal models
Analysis, Simulation
Automated Test Generation
Analysis: Cheddar, Schedulability Fault Propagation, FPTC, FIACRA
Assurance Case

Fig. 1 ADSL Approach.

tem assurance and associated evidence (e.g., artifacts from analysis tools, model checkers, theorem provers, and automated test generation) and test artifacts. Ideally, these assurance-cases can be constructed in the context of aerospace development standards (e.g., DO-178C [24], DO-254 [25], DO-297 [26], ARP-4754 [27], ARP-4761 [28]).

We have not completed this full integrated vision, but our ADSL makes significant advances toward it. In particular, we address the difficult aspect, which is developing a succinct language for specifying distributed systems with sufficient fidelity to synthesize implementations as well as formal models for verification.

Such a vision touches upon a wide range of research, which we highlight in Section II, particularly focusing on architectural description languages and formal analysis tools, our current focus. Earlier, we have described our language as an *domain-specific* language; the domain we focus on here are fault-tolerant distributed systems commonly found in aerospace systems. Toward that end, we outline the concepts that a language specialized for this domain should imbue. For example, such a language must be able to succinctly allow designers to specify and reason about different fault models, message-passing constructs, and real-time constraints. We describe these concepts in Section III. The concepts drive the design and implementation of the ADSL itself, described in Section IV. There we introduce the prototype ADSL we have built, which we call LIMA, standing for "**L**anguage for **I**ntegrated **M**odeling and **A**nalysis". We also describe our compilation strategy to both C code and to the input language of Sally, a state-of-the-art model-checker [18] that LIMA targets. In particular, the translation to Sally is particularly novel, as we describe an efficient encoding of time, distributed communication, and faults that is amenable to decidable formal analysis. To demonstrate the effectiveness of LIMA, we present case-studies drawn from aerospace systems in Section V that we model in the LIMA and generate executable source code and formal models. These case-studies highlight a unique design feature of the LIMA: it is an *embedded* domain-specific language (EDSL), meaning that it is hosted in a general-purpose programming language. The EDSL approach is common in the programming languages community; we show how it can be used to build powerful new modeling abstractions without introducing new primitives. Finally, we present conclusions and future work in Section VI.

II. Related Work

In this section, we overview related tools and approaches for modeling and verifying systems. We focus on architectural modeling languages and formal verification specialized for distributed systems, particularly noting their strengths and weaknesses with respect to specifying and reasoning about architecture, behavior, and faults in a unified way.

A. Architectural Modeling Languages

The Architecture Analysis and Design Language (AADL) was one of the first system architecture languages, evolving out of the META-H [29] language developed by Honeywell as part of the DARPA DASADA program. Since its conception, AADL has matured significantly and is now standardized by the Society of Automotive Engineers (SAE) under AS5506 [21]. AADL is primarily intended to be a system integration language, allowing generation of an integrated model that address different aspects of the system to be captured. At the core, AADL provides a common notation that supports the specification of both logical and physical aspects of the system architecture. The core language is component-based. The physical aspects of the system may be specified utilizing a extensible palette of hardware component primitives, such as processor, device, and system components, which may be interconnected with bus components and access connections. The logical notation of AADL is also component-based. AADL provides a number of software/logical model primitive abstractions, such as system, process, thread, subprograms, that allow for logical model specification. In the logical model, components are interconnected using data and event port connections. The core of AADL also allows for the association between the logical and physical models to be specified by binding property annotations to the model.

Through a flexible annex mechanism, the core AADL language can be extended with user-defined syntax. Different aspects of the system can be specified using dedicated annexes. Of specific interest are the behavior and error annexes, and the emerging constraint and hybrid annexes. The behavior annex allows for discrete behavior to be specified using a finite state machine annotation that can be associated with the logical abstraction components. This behavior may be fused with the platform behavior of the AADL core to implement a full system simulation [30]. Similarly, the error annex annotations allow probabilistic component error models and state machines to be associated with each component. Error flow and error propagation annotations are also possible to describe cross component error propagations and influences.

Using such annotations, model-based safety analyses are possible, with the annotated AADL model used as the basis for for fault-tree generation [16]. Recent annex developments include the requirements annex [31] that allows a systematic refinement and association of requirements with AADL model elements, and the hybrid annex, that intends to extend AADL to address continuous system models, and a constraint annex that allows for constraints and structural assertions to be defined and executed within the AADL modeling framework.

The System-Level Integrated Modeling (SLIM) is a simpler variant of the ADDL. It has been developed as part of the COMPASS project [10, 32]. The intent of SLIM is to generate formal architecture language that can be used as the basis of architectural analyses using formal methods.

To this end, SLIM is much simpler than AADL, excluding some of the elaborate AADL features for hierarchical abstraction and interface complexity management. SLIM also excludes some of the tasking and dispatch semantics of the underlying platform execution model. Therefore, logical abstractions, such as a periodic thread dispatch, need to be explicitly modeled using the SLIM behavioral language. However, in SLIM, behavioral and error modeling is integrated into the core model. Using a mechanism called model extension, SLIM allows these annotations to be integrated into a formal transition system model that can be used as the bases for formal analysis. Thus, SLIM supports an integrated view of nominal and error behavioral models. This differs significantly from the AADL approach, where there is little formal cross-annex linkage or semantics.

MILS-AADL, a derivative of AADL, is also under development as part of the DMILS-project[33]. This work is also targeting synthesis towards back end formal verification tooling using BIP [34]. The D-MILS project additionally targets implementation platforms based on TTEthernet.

The Robot Architecture Definition Language (RADL) [35] is a minimalist AADL targeted towards the design of multi-rate distributed systems. It is under development by SRI. Similar to SLIM, RADL is simpler than AADL, and forgoes the more elaborate features interface and property specification. RADL is also targeted towards a quasi-synchronous system architectural pattern, in which all nodes asynchronously execute tasks and exchange messages at defined periodic intervals [36]. The RADL framework also incorporates an automatic build system, *Radler*, which synthesizes glue code and platform binding code. At the time of writing, RADL does not incorporate fault modeling.

SysML [19] extends UML to address the needs of system engineering. It has been standardized under the OMG. SysML comprises a very rich palate of notations that can be utilized to specify system structure and behaviors. The notation is extendable, making it very adaptable to different modeling needs. Given a disciplined model-based system engineering approach, SysML can be used to capture the functional, logical and physical aspects of architecture, as demonstrated by Pearce and Friedenthal [37].

Through dedicated profiles, SysML notation can be extended. For example, via a Modelica profile [38], SysML can be used to model continuous systems. An automated translation is also available between Modelica and SysML. Similar translations are also under development for VHDL-

AMS [39]. SysML-AADL profiles have also been developed to support formal platform modeling [40, 41]. SysML supports relating requirements across all of the modeling elements.

Due to the flexibility of the notation, SysML can therefore be used to capture many aspects of a system archiecture using a common notation. SysML further provides a requirements framework to allow requirements to be refined via associates to modeling blocks, providing a similar capability to the AADL requirements annex discussed above.

Recent work addresses fault modeling within SysML [37], allowing SysML models to be annotated with failure modes, although this work is less mature than the AADL error annex.

The SCADE-System [42] defines an IMA modeling profile that provides a framework to define functional, logical and physical platform models within SysML. The tool also provides an extensive framework to generate Interface Control Documents (ICD) from integrated models. The tool additionally provides code generation to configured commercial partitioned operating systems and network configuration tables from the system model. The SCADE_System tool is fully integrated with the SCADE Suite, hence lower-level system behaviour can be specified in SCADE but remains linked to the higher level architecture. Such properties make this variant of SysML and interesting synthesis target for ADSL.

Matlab's Simulink [43] is one of the most widely used model-based-design notations. It provides a very rich simulation capability that allows for behavioural exploration. However, the Simulink notation lacks many of the features required for architecturally centric design; for example, the separation of logical and platform designs and the associated bindings is missing. The core language also lacks formal semantics, and is defined with reference to the behavior of the simulator. The tooling also offers limited provisions for structured design and design factoring, which may also limit its applicability to true architectural modeling. That said, many production systems have been developed using Simulink, and additional tools have been developed to broaden the applicability of Simulink. One such tool is HIPHOPS [13], that allows for fault and error annotations to be added to the Simulink models, and provides a framework to use the model as the basis of Failure Mode and Effects Analysis (FMEA) and system fault-tree analysis and generation.

B. Distributed System Modeling Languages

A variety of formal modeling and verification languages and tools have been developed specifically targeting distributed systems. Hoare's *Communicating Sequential Processes* (CSP) is one of the original and most influential distributed system process calculi [44]. CSP-based tools such as CSP$_M$ [45], JCSP [46], FSPJ [47], and CSP++ [48] have been developed and are summarized and compared in a recent report [49]. Notably, CSP++ is a relatively recent tool that includes code generation capabilities to generate C++ implementing the semantics of a specified system. Because it uses the same input language as other tools, such as FDR, a CSP-based model-checker [50], specified systems can be model-checked. However, application code must be written by-hand and spliced in. This ability is unsound insofar as application code can break invariants of the concurrency model. However, a tool like CSP++ takes promising steps in the direction of the research we present.

That said, the basic semantics do not typically handle the aspects of distributed systems with which we are concerned. For example, there are no built-in notions of faults or timed behavior. Perhaps more significantly, CSP has a dynamic model of a process, in which a process can be composed to form new processes. A more static notion of processes may be appropriate in our domain. Indeed, note the following, when trying to formalize a very simple fault-tolerant protocol in various CPS-based tools:

> As with the previous examples, our goal in this project is to use our three translation techniques on each example. The Byzantine Agreement Protocol, however, proved to be far more complex than the other examples. So complex, in fact, that the various shortcomings of each technique proved too substantial to achieve translation [49].

We present a specification of Byzantine Agreement in Section V B within our ADSL.

C. General-Purpose Formal Verification Tools

General-purpose formal verification tools have been applied extensively to the specification and verification of fault-tolerant distributed systems.

Model checkers such as the *Symbolic Analysis Laboratory* (SAL) [51], SMV [52], and the *Temporal Logic of Assertions* (TLA)'s model-checker [53] have been used to specify and verify both software and hardware distributed systems and protocols [54–59]. Model-checking is one of the most successful verification approaches for distributed systems, as the technology is “push-button” and model-checkers have become exponentially more powerful as a function over time. Tools such as SAL and the recently-developed infinite-state verification tool *Ivy* [60] allow users to supply invariants to scale verification. Still, most approaches to model-checking require ad-hoc abstractions and by-hand models. One goal of our ADSL workbench is automatic translation to model-checkers, creating sound abstractions for the user automatically.

Work in controller synthesis, usually from temporal logic specifications, has recently been applied to fault-tolerant algorithms [61]. In this work, a simple self-stabilization protocol is synthesized from an LTL specification using Boolean satisfiability. The approach uses a *counter-example-guided inductive synthesis* approach [62] to improve scalability.

Another synthesis approach is followed by Liu *et al.* with their *DistAlgo* language and tool [63, 64]. *DistAlgo* provides constructs for specifying distributed fault-tolerant algorithms embedded in a programming language, like Python. While specifications are terse, it can generate fairly efficient code, both in code size and execution efficiency [64].

Theorem-proving, in contrast to model-checking, is largely manual but quite powerful. In particular, PVS [65] has a long history of being used to specify and verify distributed systems [66, 67]. Like with model-checking, abstractions are usually ad-hoc and specifications are done by-hand. There is usually no formal correspondence with an implementation.

III. ADSL Desiderata

We first motivate the need for another architectural description language, then we present the fundamental concepts necessary for such a language. We focus on three concepts: clock models, channel & buffer models, and fault models.

A. Why Another ADL?

With the numerous architectural description languages (ADLs) and accompanying tools available (see Section II A), we must ask is is, *"Why another ADL?"*. We are focused on the correct specification and synthesis of distributed systems and the formal verification and validation of distributed system protocols. A key research goal is to provide a specification framework that encompasses a suitable level of abstraction to allow for the efficient behavioral specification of distributed fault-tolerant protocols. Additionally, we desire for protocol specifications to exist within the larger context of the system architecture and anticipated fault models.

Currently, AADL is one of the most mature ADLs, yet, in our work to date we have found that expressing the details such of protocols with AADL is non-trivial, with certain aspects not yet possible. such as behaviours that are not compatible with the underlying AADL dispatch semantics.

A second goal is integrating the models of faults and behavior. Once again, this is an area where current ADLs are lacking. For example, in AADL, the integration of the behavioral and error annexes is not mature, and hence cross-annex formal semantics and linkages are not defined. In addition, although the LIMA language (see Section II A) incorporates provisions for integrated specification, the level of abstraction is more cumbersome than it should be. This is another area where our synthesis strategy to an intermediate ADL may be beneficial. For example, using our approach, we may be able to efficiently synthesize LIMA models for formal analyses, where such models may too expensive to develop by hand.

The final intent is to link the formal assurance argument within the ADSL work flow. Once again, this is an area where the current ADLs continue to develop; the recent work with AADL and RESOLUTE [68] looks promising.

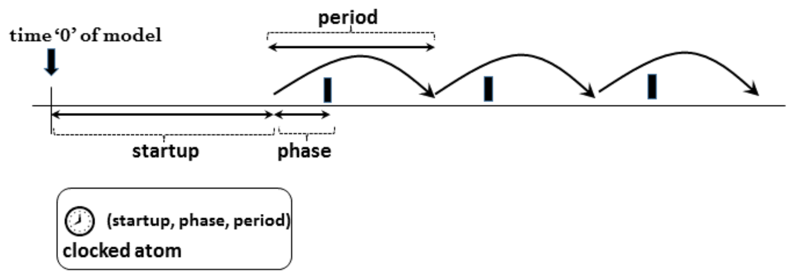
In summary, there exist no ADLs that allow the efficient specification and refinement of system models and protocol specification. Our hope is that it will be more efficient and lighteweight. That said, where formal semantics exist for intermediate ADLs, our ADSL can target them.

B. ADSL Concepts

The three fundamental concepts we present are clocks, channels & buffers, and faults to describe real-time distributed systems. We describe each in turn.

As we describe the following, we present the concepts based on the notion of an *atom.* An atom is a hierarchical state-machine. An atom can represent a node in a distributed system, but we also allow for the existence of a *sub-atom*, that is a sub-component of a node. Atoms allow us to decompose specifications. We call a sub-atom's encompassing atom the sub-atom's *parent.*

Fig. 2 Clocked Atom Model



In modeling distributed systems, we must often assign clocks to atoms to measure its notion of the passage of time. The notion of clocked nodes are *periodic* processes or tasks. A clocked atom is defined by a tuple of three parameters as shown in Figure 2:

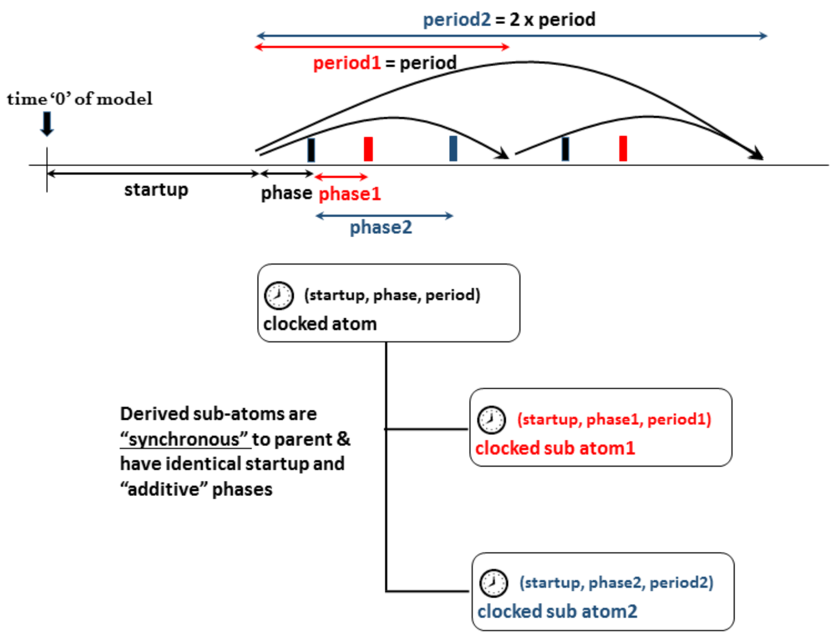
- *startup* which indicates the duration of time elapsed from time 0 when the clock starts ticking away i.e. initialized *start up delay* for the clock. Note that this startup can span multiple periods potentially. $startup \geq 0$

- *phase* is the *offset* within the period when the task/process is executed periodically. $0 \leq phase < period$

- *period* is the periodicity of the task i.e. inverse of the frequency of the task.

All *derived* clocked atoms or *sub-atoms* from parent have synchronous clock with respect to the parent clock. This means, as shown in figure 3, the *startup* of the sub-atoms are identical to parent and their clocks are initialized identical to the parent. As shown in the figure both $atom1$ and $atom2$ derived from *atom* have identical *startup* parameter. Also the *derived atoms's phases are additive* Thus $atom1$ has an effective phase $phase + phase1$ from start of period and $atom2$ has an effective phase $phase + phase2$ from the start of the period. Also derived atoms period are harmonic with respect to the parent and at equal or slower rate. For example in the figure $period1 = period$ while $period2 = 2 \times period$. Thus atom and sub-atom relationships can be used to model synchronous system whereby all distributed system nodes which are synchronous with each other can be modeled as sub-atoms with clocks under a single parent atom with a "notional" clock for the whole system. Similarly ARINC 653 [69] partitions or tasks within a single node can be modeled as sub-atoms with a single parent atom.

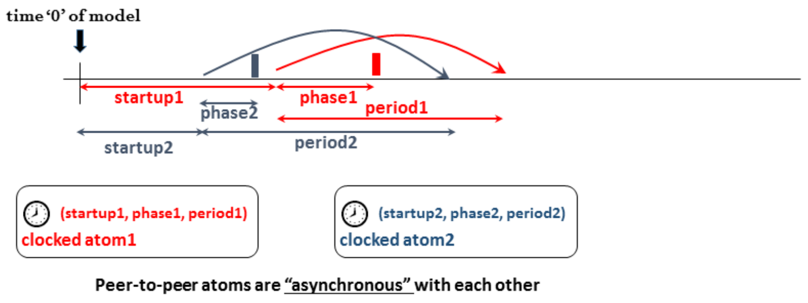
Fig. 3 Synchronous Derived Atom Clock Model



All derived sub-atoms have a synchronous clock with respect to the parent clock. That is, as shown in Figure 3, the *startup* of the sub-atoms are identical to their parent atom, and their clocks are initialized identically to the parent. A shown in the figure, both $atom1$ and $atom2$ derived from *atom* have an identical *startup* parameter. Also, the derived atoms's phases are additive. Thus, $atom1$ has an effective phase $phase + phase1$ from start of period and $atom2$ has an effective phase $phase+phase2$ from the start of the period. Finally, derived atom periods are harmonic with respect to the parent and at an equal or slower rate. For example, in the figure, $period1 = period$, while $period2 = 2 \times period$. Thus, atom and sub-atom relationships can be used to model synchronous systems whereby all distributed system nodes which are synchronous with each other can be modeled as sub-atoms with clocks under a single parent atom with a "notional" clock for the whole system. Similarly, ARINC 653 [69] partitions or tasks within a singe node can be modeled as sub-atoms with a single parent atom.

On the other hand, as shown in Figure 4, two peer clocked atoms at the top level are considered

Fig. 4 Asynchronous Peer-to-Peer Atom Clock Model


time '0' of model
startup1
phase2
phase1
period1
startup2
period2
(startup1, phase1, period1)
clocked atom1
(startup2, phase2, period2)
clocked atom2
Peer-to-peer atoms are "asynchronous" with each other

asynchronous with each other. As shown in the figure, their individual *startup*, *phase*, and *period* for their respective clocks have no relationship with each other.

During verification, it can be useful to allow startup times, periods, and phases to be nondeterministic (without violating the constraints above) to verify timing properties about the system under abstract constraints. For example, one might state timing constraints and verify that the system implements time-triggered behavior [67]. During C simulation, however, these timing values must take on constant values.

2. Channel & Buffer Model

System designers typically have to contend with managing resource constraints across networked systems. There are two high level resources they need to balance: (i) platform resources like CPU utilization (time) and memory (space) vs (ii) network resources like bandwidth/link usage manifesting as transport/channel delay (time) and network card memory/channel buffers (space). Since both platform and network resources along both time and space dimensions are all finite, optimizing along just one of those resources and/or dimension at the cost of the other is not a viable option. Correct characterization of computation time, communication time (channels delay) and associated buffers (memory at platform or network) at every node is a critical element of the ADSL as it lays the foundation for accurate modeling of platform and network resources in the system. We show the time and space attributes of network resources in terms of channel and buffer models in Figure 5.

A channel is modeled as *unidirectional* flow of message M from a transmitter Node T to multiple receiver Nodes $R_1, R_2, ..., R_n$ with corresponding channel delays $D_1, D_2, ..., D_n$. Channel delays are all different because there may be different transport paths from transmitter to the different receivers. Channel delays are a function of the size of message M, the network bandwidth/link rates, the propagation time (wire length) and the number of intermediate relays between transmitter and the receiver. Thus a message transmitted on to the channel at time T_1 from Node T is received at each of the receiver from the channel at times $T_1 + D_1, T_1 + D_2, T_1 + D_3, ..., T_1 + D_n$ respectively. Note that *unicast*, *multicast* and *broadcast* are all supported with this model of channel.

Also note that as shown in the Figure 5, there is string of *producing* processes and *consuming* processes in the model and this must correctly identified in-order and in-sequence to get the modeling of the buffer (e.g. overflow or size) correct and this is described next. For example, in Node T, there is some platform application that is producing a message (possibly from a computation process) and this produced message is stored in the channel buffer (network card memory). The network card in Node T subsequently reads from its own memory (channel buffer) and produces (transmits) the message on to the channel.

The process is then reversed at the receiver. The receiver network card consumes the message from the channel and stores the message locally into its own channel buffer (network card memory). Then either the network card reads from its channel buffer and pushes the message to consuming platform process OR the consuming platform process reads from the channel buffer and then finally processes the message as it sees fit (possibly sent to a computation process).

Since Produce/Transmission at T and Consume/Reception(s) at R can be triggered by "independent" clocked atoms (described in section III B 1) at T and $R(s)$. So buffer model is critical to manage the differences in rates and timing between production and consumption. Once the processes are correctly modeled as stated above, then we identify two types of channel buffers at either the transmitter or at the receivers:

- *Queuing*: First-in-First-Out (FIFO) Order i.e. messages are taken out of the queue in the order in which data was produced into it. The maximum size of the queue is also specified as k. If messages are not consumed at fast enough rate compared to rate at which message

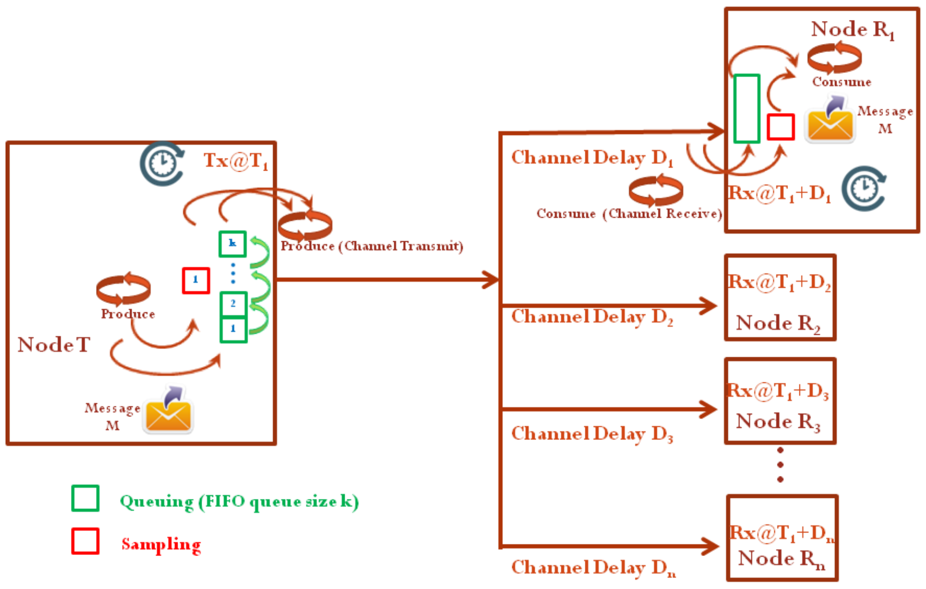


Fig. 5 Channel and Buffer Models

are produced into the buffer and once the queue is already filled with "k" messages yet to be consumed, then produced message(s) are dropped.

- *Sampling*: New message produced overwrites old message if not consumed.

3. Fault Model

The ability to model and specify fault models is a significant capability of the proposed ADSL. Faults can occur at different levels of abstraction, and an ADSL should have the capability of specifying and reasoning about faults at the different levels as well as mapping between them.

In particular, we distinguish between local and global faults. Following the taxonomy of Avizienis *et al.* [70], we refer to faults that occur locally in any component, such as a node or channel, as being modeled as *local faults*. Some examples of local faults are:

- Omission (message loss)
- Commission (babbling)
- Untimely (late, early, sequence violation etc.)

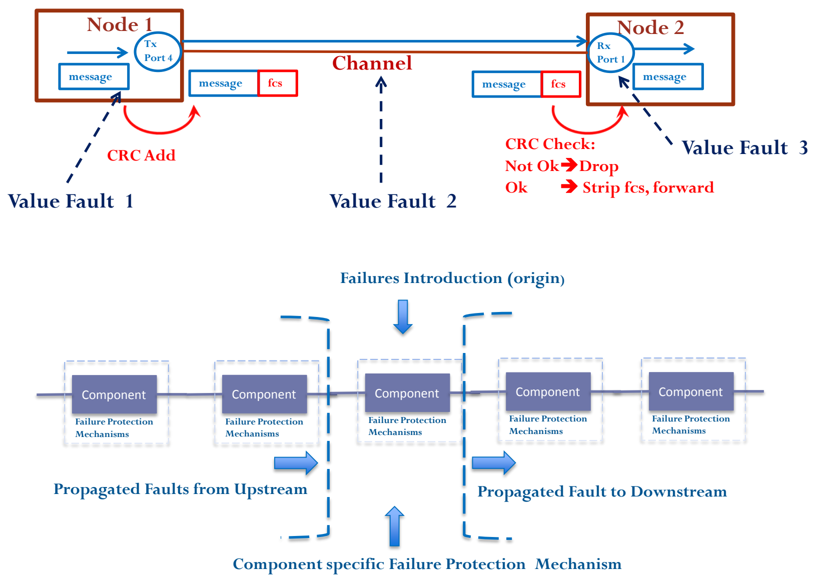


Fig. 6 Fault Propagation

- Invalid value (semantic, syntactic,..)
- Invalid protocol behavior (e.g., failure of fault handling of detection, protection etc).

On the other hand, *global faults* are based on relationships between two or more components (i.e., nodes or channels) at the system level. Examples of system faults include *symmetric* and *asymmetric* transmission faults [71]. Global faults may be dependent on the expected of degree of consensus [72], which in itself is tied to the application's sensitiveness to disagreements, independence assumptions, or degree of maliciousness (e.g. assumptions on how coordinated two or more nodes can be in triggering failures). Global faults based on relationships between two or more components can cause interactive consistency (consensus violations) and Byzantine issues at the system level.

Finally, we wish to also characterize fault propagation through the networked system, originating in some component and propagating from one component to another, as shown in the bottom of Figure 6. Faults introduced/propagated from upstream components transform to other faults based on protection mechanisms built into each component (e.g., a commission fault transforms into an omission fault if a bandwidth check is implemented as protection mechanism in a component) [73].

To illustrate this in a concrete example, refer to a cyclic redundant check (CRC) protection behavior illustrated at the top of Figure 6. Nominally in a fault-free operation, $Node_1$ adds a CRC to a message when it transmits over the channel, and when $Node_2$ receives the message, it does a CRC check by comparing the re-computed CRC with the frame check sequence (FCS). If the check fails, the message is dropped, and if the check passes, then the FCS is stripped and the message is forwarded. Note that fault-free (nominal) behavior of a CRC check at the receiver is that (i) a good message is never dropped incorrectly, (ii) a bad (corrupt) message is dropped with high probability, and (iii) there is also a small finite probability that a bad message escapes detection and drop (e.g. based on efficacy of CRC 32 etc.) [74].

Value Fault 1 in Figure 6 introduced in the node either at the indicated point or introduced upstream before that point and propagated until that point will also continue to propagate downstream and CRC offers no protection. This is because the FCS is added on an already corrupted message. Value Fault 2 introduced in the channel will be protected by CRC (modulo the CRC's efficacy). Further Value Fault 2 will transform to Omissive Fault downstream due the CRC protection. Value Fault 3 introduced in $Node_2$ at that point will propagate downstream as it is after CRC protection.

The idea then is to capture a fault transformation function in every component node/link based on behavior fault-protection schemes available at that component; e.g., a value fault transforms to an omissive fault for CRC protection. These will be specified in a static manner at every component. This way, both horizontal propagation of faults (CRC example) and vertical propagation of faults (e.g., self-checking hardware) can be modeled as a component embedded in another component) can be captured in an ADSL framework.

IV. LIMA: an ADSL Implementation

LIMA, which stands for "**L**anguage for **I**ntegrated **M**odeling and **A**nalysis", is a domain specific language embedded in the functional language Haskell [75]. Its implementation makes heavy use of Haskell as the host language for expressing macros, enabling parametrization, and the handling of parsing and low-level compilation.

The approach is an example of the embedded domain-specific language (EDSL) approach, which allows type-safe, Turing-complete compile time programming, and has been used in a number of domains, from embedded software [76] to runtime monitoring [77] to GPU programming [78]. Indeed, LIMA is built on the Atom EDSL [79]. Atom was originally designed for bare metal embedded systems programming that supported concurrency without requiring a real-time operating system. The state machine language, in which the computation of individual nodes is expressed, is within the language of Atom [80].

We first present LIMA's syntax in Section IV A. Then we describe code synthesis in LIMA in Section IV B. Formal model synthesis is presented in IV C. Finally, we briefly describe visualization tools associated with LIMA models in Section IV D.

LIMA is designed to implement the desiderata described in Section III. However, as a initial implementation, some desiderata have not been fully implemented. We describe its current status below.

A. Syntax

Our goal here is to introduce the LIMA. We are not, however, providing a comprehensive overview of the language, but we present the major elements. In particular, we elide the portion of the language focused on building state machines internal to an individual node or process. The node-local state machine language is described in the Atom itself.

We first introduce the basic building blocks of LIMA for modeling communicating state machines. Then we describe the LIMA fault model.

1. *Specifying Communicating State Machines*

There are two main syntactic elements: *atoms* and *channels*.

- *Channel*: Channels are typed and unidirectional. A channel is declared in a monadic context as follows:

```
(tx, rx) <- channel name ini
```

creating two endpoints, `tx` and `rx`, that are user-defined variables. These variables are "handles" for the channel that can be emitted or read on, respectively. The types of `tx` and `rx` must agree in all contexts. `name` is a plain text name for the channel, used to connect the syntax at this level to the formal model and C code syntax after generation. Finally, `ini` is an initial value for the contents of the channel.

- *Atom*: An atom defines a state machine that atomically handles an incoming message, updates state, and possibly emits new messages on other channels. An atom may, optionally, do this on a specified period and phase.

```haskell
myAtom rx tx = atom "myAtom" $ do
  v <- readChannel rx
  s <== value v
  writeChannel tx (Const 42)
```

The code above defines an atom named `‘‘myAtom’’` that handles a message received on a channel with a receive handle `rx`. A channel must be *dereferenced* to extract a value from it. In the declaration above, the dereferenced value, `v`, is stored into some shared-state variable `s` whose scope exceeds the current definition, and the value 42 is emitted on a channel with transmit handler `tx`. Here, we use the term "handler" to mean a function that takes an incoming message and returns an action to perform, usually doing some computation and sending out new messages.

Atoms are hierarchical, i.e. an atom may contain other atoms and thus define a hierarchy of global state variables and sub-atoms. All atoms declared within a parent atom have access to the shared state of the parent. For example,

```haskell
parent = atom "parent" $ do
  s <- int "sVar" 0
  atom "myAtom0" $ do
    ...
  atom "myAtom1" $ do
    ...
```

declares an atom named ``parent'' that contains a shared state variable named ``s'' initialized to zero. It also contains two handlers, `myAtom0` and `myAtom1`.

A typical pattern in the language is to declare a top-level atom as a container for one or more channels and one or more sub-atoms which communicate over the channels.

```
parent = atom "parent" $ do
  (tx, rx) <- channel "myChannel" 0
  s <- int "sVar" 0

  atom "myAtom0" $ do
    writeChannel tx 5

  atom "myAtom1" $ do
    cond $ fullChannel rx
    v <- readChannel rx
    s <== v
```

In this example, `myAtom0` sends the message `5` to `myAtom1` who stores it directly in the shared variable `s`. The syntax `cond $ ...` declares a guard that must be true in order for the atomic action in that block to be taken.

Finally, in addition to the hierarchical structure of shared state, the atom to sub-atom relationship extends to guards. The execution of a sub-atom is predicated not only on its guard condition, but also that of its parent, its parent's parent, etc.

2. Specifying Faults

Here we explain how we account for the fault models in LIMA, leaving the technical details of how faults are encoded in the formal model to Section IV C 3. Part of the philosophy behind our ADSL is that node behavior and fault behavior should be separate in order to make reasoning more modular. It is not a surprise then that the number and type of faults to be included in the model is not specified along with the node behavior. Instead, it is specified only in the compiler configuration.

Currently there are three options for fault model in LIMA:

- **No Faults**. All nodes in the system perform as designed and channels deliver all messages on time.
- **Fixed Faults**. A mapping from node name to fault type is given, allowing the designer to specify statically the nature of faults to be considered.
- **Hybrid Fault Model**. The designer specifies numerical weights for each of three types of fault: manifest, symmetric, and byzantine. In this mode, the model-checker will explore all possible configurations of nodes with different faults as long as the weighted sum of node-fault combinations doesn't exceed a given total.

The last option is the most powerful form of reasoning about faults in the system that LIMA offers. It generalizes the well-known results of Park and Thambidari [71] and Lincoln and Rushby [81].

The fault model is decomposed from LIMA's atom's and channel model to specify communicating state machines. This is because we need to be able to synthesize executing code from the system; the fault model is used only within a formal verification. Thus, the fault model is presented as additional configuration data about the environment. The configuration is passed to the Sally model-checker compiler.

B. Code Synthesis

In the LIMA framework we can translate specifications into executable implementations by generating C code. The code generator is optimized for embedded targets, those running without a real-time operating system. Although, the code also compiles and runs on POSIX systems as well.

At a high level, the code generator collects all the atomic actions given in a specification and translates them into C functions which are executed on a fixed, deterministic schedule. The schedule is determined by a scheduler which takes into account each atom's preferred period and phase. The scheduler also computes the number of basic expressions present in each atomic action so that the

```
ex4 :: Atom ()
ex4 = atom "ex4" $ do
  x <- int64 "x" 0
  y <- int64 "y" 0

  clocked 2 0 $ atom "atomX" $ do
    incr x
    decr y

  clocked 5 3 $ atom "atomY" $ do
    incr y

  assert "y not positive" (value y <=. 0)
```

Fig. 7 Example specification with two periodic atoms

real-time per tick can be calibrated. In section IV C we discuss synthesizing formal models from LIMA specifications. By contrast, in the formal models the schedule is left non-deterministic but fixed in each system trace.

As an example, consider the specification in Figure 7. Two shared variables are declared and two periodic atoms update their values. The code generator produces a C structure representing the global system state; in this case it consists of two integers. The fields of the struct are nested in several name spaces designed to keep local variables in different atoms from conflicting. The two atomic actions are translated into "rules"; C functions that are called by a main driver function. Figure 9 shows the rule generated for `atomX`. The code is mostly generated in static single-assignment (SSA) form, with the exception of a small number of state variables.

Finally, the rule functions are driven by a scheduling function that is responsible for maintaining a global clock value, executing the rules in the correct order and time, and executing assertion statements. This is shown in Figure 10.

The final piece of C code generation is a user supplied `main` function which should repeatedly

```
struct {  /* state */
  struct {  /* ex4 */
    struct {  /* ex4 */
      int64_t x;
      int64_t y;
    } ex4;
  } ex4;
} state;
```

Fig. 8 Global state structure in the generated C code

```
/* Rule { 0, ex4.ex4.atomX } */
static void __r0() {
  bool __0 = true;
  int64_t __1 = state.ex4.ex4.x;
  int64_t __2 = 1LL;
  int64_t __3 = __1 + __2;
  int64_t __4 = state.ex4.ex4.y;
  int64_t __5 = __4 - __2;
  state.ex4.ex4.x = __3;
  state.ex4.ex4.y = __5;
}
```

Fig. 9 Translated rule for `atomX`

call the generated scheduler function. This is typically a tight loop with a call to the scheduler followed by a delay statement that should be calibrated for the platform and desired amount of real-time per system tick.

Auto-generated code like that shown in Figure 10 is only meant to be machine readable, not human readable. One should think of this code as an intermediate representation on the way to native machine code. It can be used by human designers for debugging purposes if really needed, but the LIMA language and complilation tool chain is setup to avoid exposing users to these low

```
void ex4()
{
  {
    static uint8_t __scheduling_clock = 0;
    if (__scheduling_clock == 0) {
      __assertion_checks(); __r0();  /* Rule { 0, ex4.ex4.atomX } */
      __scheduling_clock = 1;
    }
    else {
      __scheduling_clock = __scheduling_clock - 1;
    }
  }
  {
    static uint8_t __scheduling_clock = 3;
    if (__scheduling_clock == 0) {
      __assertion_checks(); __r1();  /* Rule { 1, ex4.ex4.atomY } */
      __scheduling_clock = 4;
    }
    else {
      __scheduling_clock = __scheduling_clock - 1;
    }
  }
  __global_clock = __global_clock + 1;
}
```

Fig. 10 Generated scheduler function

level details.

Certification (e.g., DO178-C [24]) of auto-generated code has been addressed elsewhere for tools such as Simulink and Stateflow [82]. Of course, such an approach requires LIMA undergo tool qualification.

Execution of the generated code can be very useful in the design and debugging of systems in

```haskell
ex4 :: Atom ()
ex4 = atom "ex4" $ do
  x <- int64 "x" 0
  y <- int64 "y" 0

  clocked 2 0 $ atom "atomX" $ do
    incr x
    decr y
    probe "x + y" (value x + value y)

  clocked 5 3 $ atom "atomY" $ do
    incr y
    probe "y" (value y)

  assert "y not positive" (value y <=. 0)
 mapM_ printProbe =<< probes
```

Fig. 11 Probing the values of two runtime expressions

LIMA. To that end LIMA provides several debugging features that can be used along with forward execution in order to examine the state of the system as it evolves over time. A special primitive function called `probe` allows the user to setup a hook which monitors the value of any expression at any point of a specification. These probes can be printed out as part of the system execution by calling another special function `printProbe`. In Figure 11, the running example has been modified to include probes on certain runtime expressions and to print the values of those expressions out on every tick. The output generated during the example's execution is shown in Figure 12.

C. Formal Model Synthesis

In this section, we describe our approach for mapping the syntax and semantics of LIMA to a formal transition system model suitable for model checking. (Translation to C code is relatively straightforward and we omit its description here.) One of the main arrows of Figure 1 points from

```
y: -1
x + y: 0
y: -1
x + y: 0
y: -2
x + y: 0
y: -1
x + y: 1
y: -2
x + y: 1
y: -2
x + y: 1
y: -3
```

Fig. 12 Printing probes during execution

ADSL to "formal models". In the following sections we describe concretely how a system specified in LIMA can be translated into a model-checker, while preserving important semantic constraints.

Efficiently translating to a model-checking system without relying on ad-hoc, problem-specific abstractions to make the translation feasible is an open research problem that we focus on herein.

Let us return to our handler example from the previous section. Execution of the handler updates the state of the system and so in our translation it is represented by a transition relation. In Sally, transition relations are specified by predicates over the "current" and "next" states of the system. These are denoted by prefixing state variables with the namespaces "state" and "next".

```
handler rx tx = atom "someHandler" $ do
  let v = readChannel rx
  s <== v
  writeChannel tx (Const 42)
```

Fig. 13 Handler in the ADSL

The variable `cal` in Figure 14 is a state variable referencing the "calendar", a data structure which

```
;; someHndler:
(and msg_pending(state.cal, state.rx)
   (= next.s            msg_read(state.cal, state.rx))
   (= next.cal          msg_send(state.cal, state.tx, 42))
   (= next.other_var1   state.other_var1)
   (= next.other_var2   state.other_var2)
   ... )
```

Fig. 14 Handler as part of a formal model

keeps track of the times that future events will take place. Sending a message is implemented by adding a (future) time, channel identifier, and message content to the calendar. At the proper time, a transition is enabled for the receiver to act upon the message. This mechanism is described in more detail below. In the Sally transition relation, the notations `msg_pending(...)`, `msg_read(...)`, and `msg_send(...)` are shorthand for more complicated expressions:

- `msg_pending` is a boolean expression that checks whether there is a message available for delivery at the current time.
- `msg_read` returns a message value from the appropriate entry in the calendar.
- `msg_send(...)` is an calendar-valued expression which computes the new value of the calendar, typically involving the overwrite of a calendar entry with the message contents and the overwrite of the entry's delivery time with the current time plus the configured message delay.

Various kinds of systems with real-time constraints can be implemented on top of the calendar automata framework. In LIMA, two main constructs are used to express features such as periodic execution and aperiod timeouts. First, there is a primitive function `clocked`, which takes a concrete period value (in ticks) and either a concrete phase value (between 0 and the period), or a special value that indicates phase should be indeterminate. In this context, indeterminate phase means that phase of execution is non-deterministic, but fixed within each system trace. This allows us to explore phase-dependent properties in real-time systems such as the Automatic Airbreak System case study presented in V C.

Second, there is another primitive function `writeChannelWithDelay` which, like `writeChannel`, sends a message over a channel, but with a specified delay added to the delivery time. This feature can be used to program reset-able, aperiodic timeouts. For example, in the following specification, a node sets itself a timeout of 100 ticks, after which it sets a flag.

```
timeout = atom "timeout" $ do
  flag <- bool "flag" False
  (tx, rx) <- channel "self_loop" False
  writeChannelWithDelay 100 tx true

  atom "on_wake" $ do
    cond $ fullChannel rx
    flag <== true
```

In this example, the relationship between the outer atom and the inner atom is key. The `flag` and `channel` are shared between the two atoms (being in scope for both of them), but the "on_wake" atom's execution is predicated on the guard of both itself, and its parent. Since the parent has no guard in this case we can think of "timeout" and "on_wake" as two different nodes communicating over the "self_loop" channel.

1. *Calendar Automata*

Real-time system verification in general-purpose model-checkers requires an explicit formalism of real-time progression. Trying to encode real-time clocks directly is difficult; in particular, one must avoid Zeno's paradox in which no progress is made because state transitions simply update real-valued variables by an infinite sequence of decreasing amounts whose sum is finite. To avoid this problem, Dutetre and Sorea developed *calendar automata* [59], which is itself inspired by event calendars used in discrete-event simulation. Rather than encoding "how much time has passed since the last event", it encodes "how far into the future is the next scheduled event", and a real-valued variable representing the current time is updated to the next event time.

Define a set of *events* $e_0, e_1, \ldots, e_n \in E$. For now, we do not define events; intuitively, an event is a set of state variables (shortly, we will associate events with messages sent in a distributed system). When an event is *enabled*, the transitions over events are enabled; otherwise, the variables stutter (maintain the same value).

An *event calendar* $\{(e_0, t_0), (e_1, t_1), \ldots, (e_n, t_n)\}$ is a set of ordered pairs (e_i, t_i) called *calendar events* where $e_i \in E$ is an event and $t_i \in \mathbb{R}$ is a *timeout*, the time at which the event is scheduled. We denote element (e_i, t_i) of an event calendar by c_i.

Let cal be an event calendar and $c_i, c_j \in cal$ be calendar events. Define an ordering on calendar events such that $c_i \leq c_j$ iff $t_i \leq t_j$, and $\min(cal) = \{c_i | \forall c_j \in cal,\, c_i \leq c_j\}$ are the minimum elements of cal.

Let a transition system $\mathcal{M} = (S, I, \rightarrow)$, be a set of states S, a set of initial states $I \subseteq S$, and a transition relation $\rightarrow \subseteq S \times S$. We implicitly assume a set of state variables such that each state $\sigma \in S$ is a total function that maps state variables to values. We sometimes prime a state to denote that it satisfies the transition relation: $\sigma \rightarrow \sigma'$. We also sometimes use a variable assignment notation to describe what state variables are specifically updated: e.g., $\sigma' = \sigma[v := v + 1]$.

We distinguish two special state variables in a transition system: (1) $now \in \mathbb{R}$ denotes the current time in the state, and (2) cal is an event calendar.

The following laws must hold of a transition system $\mathcal{M}$ implementing a calendar automaton:

1. Time is initialized to be less than or equal to every calendar timeout: $\forall \sigma \in I$, $\forall (e_i, t_i) \in \sigma(cal)$, $\sigma(now) \leq t_i$.

2. In all states, if the current time is strictly less than every calendar event, then the only enabled transition is a *time progress* update: $\forall \sigma \in S$, $\forall (e_i, t_i) \in \sigma(cal)$, if $\sigma(now) < t_i$, then $\forall \sigma'$ such that $\sigma \rightarrow \sigma'$, $\sigma' = \sigma[now := \min(cal)]$.

3. In all states, if the current time equals a timeout, then the only transitions enabled are calendar event updates associated with the timeout: $\forall \sigma \in S$, $\exists (e_i, t_i) \in \sigma(cal)$ such that $\sigma(now) = t_i$ implies $\forall \sigma'$ such that $\sigma \rightarrow \sigma'$, $\sigma'(now) = \sigma(now)$, $\sigma'(c_j) = \sigma(c_j)$ for all $c_j \in \sigma(cal)$ such that $c_j \neq c_i$ (recalling that by convention, $c_i = (e_i, t_i)$), and $c_i \notin \sigma'(cal)$.

From the definitions, it follows that in every state, the timeouts are never in the past, and that time is monotonic:

Lemma IV.1 (Future timeouts) *$\forall \sigma \in S$, $(e_i, t_i) \in \sigma(cal)$, $\sigma(now) \leq t_i$.*

Lemma IV.2 (Monotonic time) *$\forall \sigma, \sigma' \in S$, if $\sigma \rightarrow \sigma'$, then $\sigma'(now) \geq \sigma(now)$.*

Proofs of these two lemmas are straightforward and omitted.

In a distributed system, it is convenient to distinguish global actions and local actions. Global actions are principally interprocess communication, while local actions are those carried out by each process to update its local state and produce new messages to broadcast. While both global and local actions can both be modeled as events in a calendar automata, doing so is generally overkill and complicates the model. From the global perspective, individual processes can update their local state atomically.

Again, following Dutetre and Sorea, we associate calendar events with channels in a distributed system [59]. Specializing calendars to message passing does not lose generality since all external communication from an individual process can be abstracted as message passing. Furthermore, fault models can be abstracted to act over channels rather than processes [83]. The calendar introduces real-time constraints on when processes send and receive messages.

Assume processes are indexed from a finite set Id. A *channel* from process i to j is an ordered pair (i, j). Fix a set of messages Msg. Given a channel and a timeout, let *send* be a relation on messages sent on a channel at a given time:

$$send \subseteq \text{Id} \times \text{Id} \times \mathbb{R} \times \text{Msg}$$

So $send(i, j, t, m)$ holds iff i sends to j message m at time t. Likewise, let

$$recv \subseteq \text{Id} \times \text{Id} \times \mathbb{R} \times \text{Msg}$$

be a relation on messages received on a channel at a time, so that $recv(i, j, t, m)$ holds iff the message m received by j from i at time t.

In the absence of faults, we require that messages received were previously sent and not previously received: if $(i, j, t, m) \in recv$, then $\exists t'$ such that $(i, j, t', m) \in recv$ where $t' < t$, and $\neg\exists t''$ such that $t' < t'' < t$ and $(i, j, t'', m) = (i, j, t, m)$. (We address faults in Section IV C 3.)

Then an event calendar for sending and receiving messages on channels is the union of the *send* and *recv* relations.

The event of receiving a message initiates a process to update its local transition system and generate additional messages to send. When the process is updating its local transition system, the event calendar is paused. That is, updating an event $(i, j, t, m) \in recv$ also includes updating j's transition system.

2. System Properties

In order to specify safety properties of a system in our ADSL, we've chosen to use the synchronous observer pattern [84]. The synchronous observer is a module that is composed with a target system synchronously. Its job is to monitor the state variables of the system and raise a flag when safety properties are violated. This idea is a popular alternative to specifying properties in a special language (usually a temporal logic like LTL) that has the advantage of being written in the same language and environment as the system. Additionally, writing a synchronous observer is second nature to the systems engineers; it is essentially like adding a runtime check to a program.

Using observers allows us to sidestep the integration and expression of temporal logic in our ADSL. Instead, the programmer adds an observer to her system as a top-level monitor along with annotations that indicate which state variables are to be observed. From this we automatically generate a corresponding observer module as well as the theorems and lemmas that the model checker expects.

Our experiments with a few specific distributed fault-tolerant systems suggest that using synchronous observers in place of LTL properties introduces very little overhead during verification.

3. Fault Models

The typical approach to modeling faults is to add new state variables to each process representing its fault state. Then a node chooses actions based on its fault state. As a simple example, we might define a node that sends a good message if it is non-faulty and a bad message otherwise. In pseudocode using guarded commands, its definition might look like the following:

```
node:
  health: Fault_Type;
  faulty(health)     --> send(bad_msg);
  non_faulty(health) --> send(good_msg);
```

But this approach mixes the specification of a node's behavior with the fault model, an aspect of the environment. Generally, nodes do not contain state variables assigned to their faults, or use their fault-status to determine their behavior [85]! The upshot is that combining faults and node state divorces the specification from its implementation.

A second difficulty with model-checking fault-tolerant systems in general is that modeling faults requires adding state and non-determinism. The minimum number of additional states that must be introduced may depend non-obviously on other aspects of the fault model, specific protocol, and system size. Such constraints lead to "meta-model" reasoning, such as the following, in which Rushby describes the number of data values that a particular protocol model must include to model the full range of Byzantine faults (defined later in this section):

> To achieve the full range of faulty behaviors, it seems that a faulty source should be able to send a *different* incorrect value to each relay, and this requires n different values. It might seem that we need some additional incorrect values so that faulty relays can exhibit their full range of behaviors. It would certainly be safe to introduce additional values for this purpose, but the performance of model checking is very sensitive to the size of the state space, so there is a countervailing argument against introducing additional values. A little thought will show that .... Hence, we decide against further extension to the range of values [54].

The second problem is the most straightforward to solve. In infinite-state model-checking, we can use either the integers or the reals as the datatype for values. Fault-tolerant voting schemes, such as a majority vote or mid-value selection (see Section V B), require only equality, or a total order, respectively, to be defined for the data.

The solution to the first problem is more involved. Our solution is to introduce what we call a *synchronous kibitzer* that symbolically injects faults into the model. The kibitzer decomposes the state and transitions associated with the fault model from the system itself. For the sake of concreteness in describing the synchronous kibitzer, we focus on a particular fault model, the hybrid fault model of Thambidurai and Park [71]. This fault model distinguishes Byzantine, symmetric, and manifest faults. It applies to broadcast systems in which a process is expected to broadcast the same value to multiple receivers. A *Byzantine* (or *arbitrary*) fault is one in which a process that is intended to broadcast the same value to other processes may instead broadcast arbitrary values to different receivers (including no value or the correct value). A *symmetric* fault is one in which a process may broadcast the same, but incorrect, value to other processes. Finally, a *manifest* (or *benign*) fault is one in which a process's broadcast fault is detectable by the receivers; e.g., by performing a cyclic redundancy check (CRC) or because the value arrives outside of a predetermined window.

Define a set of fault types

$$\text{Faults} = \{none, byz, sym, man\}.$$

As in the previous section, let Id be a finite set of process indices, and let the variable

$$faults : \text{Id} \rightarrow \text{Faults}$$

range over possible mappings from processes to faults.

The hybrid fault model assumes a broadcast model of communication. Define $rnd : \mathbb{R} \rightarrow \mathbb{N}$ such that if $rnd(t_0) < rnd(t_1)$, then $t_0 < t_1$. A $broadcast : \text{Id} \rightarrow 2^{\text{Id}} \rightarrow \mathbb{R} \rightarrow \text{Msg} \rightarrow 2^{E}$ takes a sender, a set of receivers, a real-time, and a message to send each receiver, and returns a set of calendar events:

$$broadcast(i, R, t, m) = \{m | j \in R \text{ and } send(i, j, t) = m\}$$

With this machinery, we can define the semantics of faults by constraining the relationship between a message broadcast and the values received by the recipients. For a nonfaulty process that broadcasts, every recipient receives the sent message, and for symmetric faults, there is no requirement that the messages sent are the ones received, only that every recipient receives the same value:

$$\begin{array}{l} nonfaulty\_constraint = \\ \quad \forall i, j \in \text{Id}, t \in \mathbb{R}. \\ \qquad faults(i) = none \\ \qquad \text{implies } recv(i,j,t) = send(i,j,t) \end{array}$$

$$\begin{array}{l} sym\_constraint = \\ \quad \forall i, j, k \in \text{Id}, t \in \mathbb{R}. \\ \qquad (\quad faults(i) = sym \\ \qquad \text{and } broadcast(i, \{j,k\}, t, m)) \\ \qquad \text{implies } recv(i,j,t) = recv(i,k,t) \end{array}$$

Byzantine faults are left completely unconstrained.

Thus, faults can be modeled solely in terms of their effects on sending and receiving messages. A node's specification does not have to depend on its fault status directly.

Implementing the synchronous kibitzer fault injection in Sally consists of three details. First, state variables are allocated for each of the system nodes to represent its fault state. A state variable is also allocated for each channel to represent potential faulty values sent over that channel. These channel values are left to vary non-deterministically through each system trace, but they are constrained according the the fault type of the sender. For example, if the sender is symmetrically faulty, then the fault values of all its outgoing channels should be non-deterministic, but equal. Second, these state variables are constrained by fault model formulas generated by the compiler depending on the fault model chosen in configuration. In the general Hybrid Fault Model case, these formulas are inequalities involving weighted sums over the nodes. Last, the expression denoted by `msg_read` in subsection IV C is a conditional. If the sending node's fault state is non-faulty, then `msg_read` returns the intended message (a value that lives in some entry on the calendar). If not, then `msg_read` returns the (constrained) non-deterministic fault value associated with the sender.

D. Visualization

We conclude this section by mentioning a third backend for LIMA that allows system specifications to be visualized in various ways. To produce a visualization one calls the `graphAtom` function

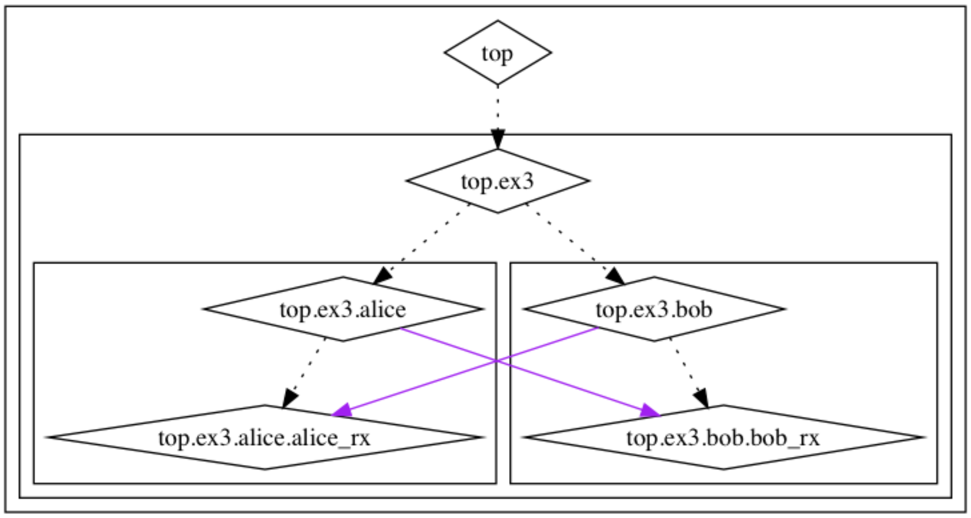


Fig. 15 Graph representation of an atom specification

with a filename prefix and an `atom` specification. The result is a `png` image file containing an abstract representation of the system. Nodes are displayed with their name in diamonds. Atom/Subatom relationships are denoted with dotted black arrows and channels are depicted by purple arrows. Figure 15 shows a typical example.

V. Case-Studies: Representative Distributed Systems

As mentioned in Section I, our ADSL is focused on the kinds of systems found in avionics. Particularly, we limit ourselves to distributed systems with the following characteristics: a finite, fixed, and usually small number of nodes; a fixed number of communication channels between nodes; possibly local and system-wide real-time constraints; and fault-tolerance requirements and constraints.

We present here representative case-studies, each highlighting particular aspects that the ADSL must handle. These case studies exercise and demonstrate the breadth and expressiveness of our ADSL:

- a high level model of a redundant, switched ethernet network
- the Hybrid Oral Messages protocol [81], OMH(1), which is a *synchronous* Byzantine agreement protocol supporting a hybrid fault model;

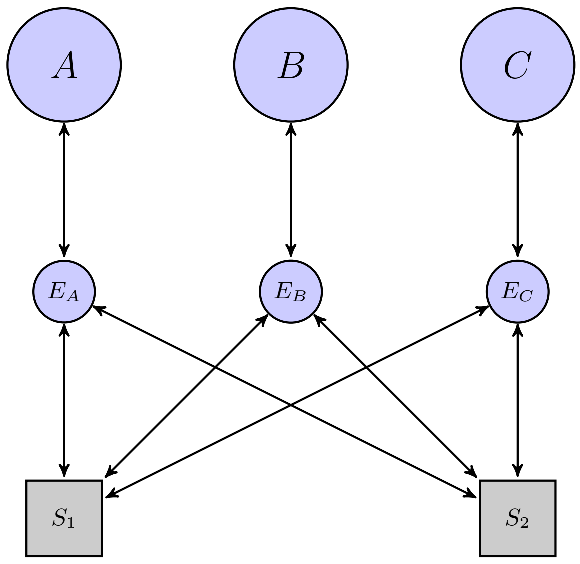


Fig. 16 A switched ethernet network for 3 nodes and 2 switches

- and an *asynchronous* "push-button" brake-by-wire [86] case-study.

For each case-study, we first informally describe the protocol or system, then we present its formalization in the ADSL. We focus on how the primitives and library functions built on those primitives succinctly capture the model.

A. Switched Ethernet Network

As a warmup to the more complicated case-studies presented in later sections, we start with a high-level model of a redundant, switched ethernet network. This network provides broadcast communication among a set of n nodes and provides redundancy through a set of m independent switches. Figure 16 depicts the network for the choice of 3 nodes and 2 switches.

The network operates as follows. A node, say A, wants to broadcast a message. It sends a message to an endpoint node E_A that handles the broadcast to each of the 2 switches S_1 and S_2. When a switch receives a message it relays it to all the other endpoints on the network. The endpoints are responsible for sorting out which message to eventually deliver to the node. For simplicity we describe a network where endpoints deliver all messages they receive from switches to their corresponding node.

To specify a network in LIMA we declare a function `mkSWEther` that takes as parameters a number of nodes and a number of switches and returns a list of channel input/output pairs. The

```
-- generate the internal channels: [ [ (in_k_j, [out_1, ...]) ] ]
-- where in_k_j goes from endpoint j to switch k and out_1 .. out_{n-1} go
-- from switch k to the other endpoints (but not the j-th).
internalChans <-
  forM rm # \k ->        -- loop over switches
    forM rn # \j -> do   -- loop over endpoints
      in_k_j <- channel (printf "in_s%d_e%d" k j) typ
      let mkOChan i = do c <- channel (printf "out_s%v_e%v_e%v" k j i) typ
                         return (i,c)
      outs <- mapM mkOChan (bar j)
      return (in_k_j, outs)
```

Fig. 17 Declaration of internal channels

channels are meant to be attached to user specified nodes in a specific order. Internally, `mkSWEther` builds the endpoints and switches as nodes and builds all the channels in between as well as those pointing in and out of the network which will be returned.

Figure 17 shows how the internal channels are built. There is a unidirectional channel for each switch and each endpoint and each other endpoint. They are stored in a particular order for ease of use in attaching them to switches and endpoints.

The switches are built from atoms having n handler sub-atoms. Each handler listens to a particular incoming channel (from one of the endpoints) and whenever it sees a message there it broadcasts it out to all the other endpoints. The declaration of switches is seen in Figure 18.

The endpoints are mostly similar to the switches: they listen to the channel coming from the corresponding node and broadcast and received message to the switches. However, in the opposite direction we have a problem. Each endpoint must also listen to all the switches and decide what to do with the messages. Listening is not a problem, we simply declare sub-atoms for each incoming switch channel and set them up to listen to the correct channel. But now we need these sub-atoms to write each message they receive to the one outgoing channel that points to the corresponding node. This is a problem because it means we have multiple atoms writing to the same channel. This

```
-- generate the switches:
-- each one listes on each incoming chan and broadcast to all outgoing chans
forM_ rm # \k ->
  atom (printf "sw%v" k) # do
    let myChans = internalChans !! k  -- :: [ (in_k_j, outs) ]_j
    forM_ rn # \j -> do
      let (myIn, myOuts) = myChans !! j
      atom (printf "handler_%v_%v" k j) # do
        cond # fullChannel (snd myIn)
        v <- readChannel (snd myIn)
        mapM_ ((`writeChannel` (v :: E Typ)) . fst . snd) myOuts
```

Fig. 18 Declaration of switches

is not allowed in LIMA by design. Instead we must buffer the sending of messages to the node. In our case-study implementation we chose to buffer messages using a FIFO queue of fixed length.

B. Synchronous Fault-Tolerant OM(1) Systems

1. *Informal Model*

Our first case study is is a system that implements one of the "Oral Messages" algorithms. These are synchronous, distributed, and fault-tolerant systems that solve the Byzantine Generals Problem [87]. The family of systems that implement OMm have different numbers of communicating nodes and offer varying levels of fault tolerance. We've chosen to focus our attention on systems that implement the specific algorithm OMH1, "Hybrid Oral Messages with 1 Round". In this algorithm, n nodes communicate in order to reach agreement on a valid course of action (or equivalent information). This is done in the presence of *at most 1* faulty node, whose communications and behavior are assumed to be completely unconstrained ("Byzantine"). The difference between OM(1) and OMH(1) lies in the details of how certain types of faulty messages are treated. OMH(1) can be viewed as an extension of OM(1) which tolerates a broader set of fault patterns.

Figure 19 depicts the communication pattern for OM1 (and also OMH(1)). Each of the n

nodes in the system represents a (Byzantine) General. In this formulation, the center node is the commanding general, while the $n-1$ outer generals are the lieutenants. The algorithm starts with the commanding general sending each lieutenant the same message v. Upon receipt of the commander's message, each lieutenant in turn sends the message received to each of the other lieutenants. Once a lieutenant has received all $n-1$ expected messages, a majority vote is taken among the n messages and the lieutenant declares its output (course of action) to be whichever message is in the majority.

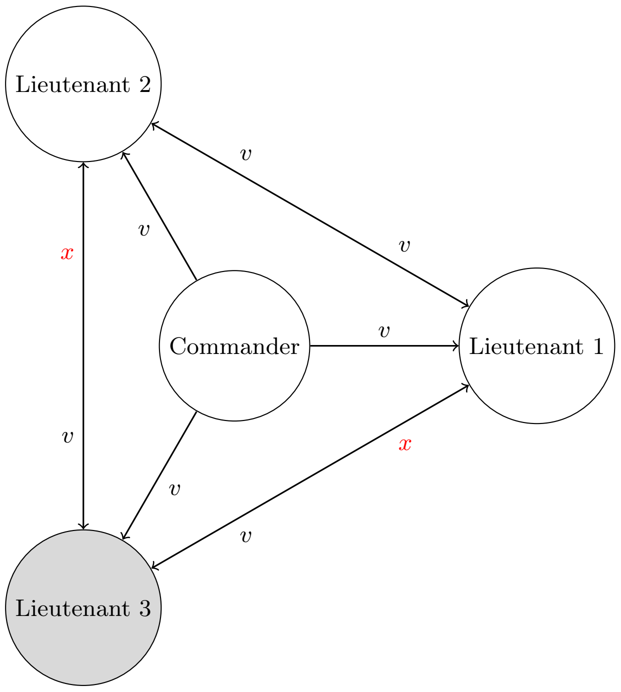


Fig. 19 OMH**1 Four generals, one traitor**

For a fault model we follow [54] and assume that nodes are the only source of faults. Moreover, node faults are assumed to be either *manifest*, *symmetric*, or *byzantine*. A fault is called manifest if it is detectable by the non-faulty nodes; for example a node that sends to another node a message that explicitly indicates there is a fault. On the other hand, a fault is called symmetric if its presence is not necessarily detectable as a fault to the other components; for example a node which sends wrong messages, as opposed to invalid or missing messages.

Given a formal model of OMH1, the standard target for verification is *validity* and *agreement*. If we let l_i denote the output for lieutenant i (where $1 \leq i \leq n-1$), then validity states:

$$\forall i. \quad l_i = v \tag{1}$$

whereas agreement states:

$$\forall\, i, j. \quad l_i = l_j \tag{2}$$

where i, j range over the non-faulty nodes.

It is a classical result that OM1 can tolerate at most 1 (byzantine) faulty node as long as there are at least 4 nodes in total. By adding additional communication rounds, OM1 can be extended to an algorithm OMm which tolerates at most m faults. On the other hand it is known that any system which tolerates m byzantine faults must involve at least $3m + 1$ nodes.

In the results above there are several assumptions made regarding the underlying computational platform which aren't obvious from the informal description, but become so when one considers formally modelling it. To make as many assumptions as possible explicit, we require:

1. every message sent by a node is received successfully by the addressed node,
2. when a node receives a message it may determine who sent it,
3. each node can detect the absence of a message.

These are the assumptions made by Lamport in his constructed solution for OMm and the corresponding impossibility result (see [87] §2). It follows that a system implementing OM1 must transition synchronously. In particular, in an asynchronous system there is no general method for detecting when a message is absent.

2. ADSL Specification

We now describe our implementation of Oral Messages in the ADSL. While we have specified and verified the extension, OMH(1) in the ADSL, we elide for simplicity the differences between OMH(1) and OM(1) and just present OM(1) here.

Our ADSL implementation of OM(1) makes heavy use of the facilities of the host language, Haskell. Indeed, the main part of the definition of the system is given in just a few lines thanks to the use of parameterization.

```
om1 :: Atom ()
om1 = do
  -- setup channels for communication between source, relays, and receivers
  s2rs <- mapM newChannel ["s2r" ++ show i | i <- relaySet]
  r2rs <- mapM (mapM newChannel) [["r2r" ++ show i ++ show j | j <- recvSet]
                                                               | i <- relaySet]
  -- declare the set of vote variables that receivers will populate
  votes <- mapM msgVar ["vote" ++ show j | j <- recvSet]

  -- declare source node
  source (map fst s2rs)

  -- declare relay nodes
  forM_ relaySet $ \idt ->
    relay ident (snd (s2rs !! idt))
                (map fst (r2rs !! idt))

  -- declare receiver nodes
  dones <- forM recvSet $ \idt ->
    recv idt [snd ((r2rs !! i) !! idt) | i <- relaySet] (votes !! idt)

  -- state the "agreement" property
  let votesEqual (v,w) = value v ==. value w
  assert "agreement" $ imply (and_ (map value dones))
                             (all_ votesEqual
                                   [(v,w) | v <- votes, w <- votes])

  -- state the "validity" property
  let voteGood v = value v ==. goodMsg
  assert "validity" $ imply (and_ (map value dones)) (all_ voteGood votes)
```

Fig. 20 Setting up OM(**1**)

```
recv :: Int             -- ^ receiver id
     -> [ChanOutput]    -- ^ channels from relays
     -> V MsgType       -- ^ vote variable
     -> Atom (V Bool)
relay idt inC outCs = atom ("relay" ++ idt) # do
  -- declare local variables
  done <- bool "done" False
  msg  <- msgVar ("relay_msg" ++ idt)

  -- activation condition:
  --   we haven't stored a value yet and there is a message waiting
  --   on the channel 'inC'
  cond # isMissing msg &&. fullChannel inC

  -- behavior
  m <- readChannel inC
  msg  <== m
  done <== true
  forM_ outCs # \c -> writeChannel c m
```

Fig. 21 Function for declaring a generic relay

In this specification we refer to the Commander as the "source" and the Lieutenants are unrolled into two sets: "relays" and "receivers". Here, the source broadcasts to the relays which, in turn, broadcast to the receivers which proceed with their vote. The code of Figure 20 shows first channels being setup for communication, then the source is declared by calling a function which we define shortly. The relays and receivers are also declared by calling functions provided different parameters.

For example, Figure 21 shows the function definition for a generic relay.

The `relay` function takes an identifier, an incoming channel (for receiving) and a list of channel outputs (for broadcasting). Recall that the `cond` primitive acts as a guard on the atomic action to be taken in the last 4 lines of the definition. In those last lines, the relay reads a message from the

```
computeVote :: [E MsgType] -> E MsgType
computeVote = fst . foldr iter (missingMsgValueE, 0)
  where
    iter x (y, c) = ( mux (x ==. y) onTrue1 onFalse1
                    , mux (x ==. y) onTrue2 onFalse2)
      where
        onTrue1      = y
        onTrue2      = c + 1
        onFalse1     = mux (c ==. 0) x y
        onFalse2     = mux (c ==. 0) 1 (c - 1)
```

Fig. 22 Fast Majority Vote in LIMA

incoming channel, stores it in a local variable, sets its `done` flag, and then writes that message out on each of the outgoing channels it has.

The definition of `recv` is similar, except that each receiver possibly makes two atomic actions through the declaration of two sub-atoms. The first sub-atom listens for messages from the relays and fills a buffer as they arrive. The second sub-atom is enabled only when the buffer is full and it computes a majority vote on the buffer and stores the result in its `vote` variable argument.

The majority vote computation is specified using the "Fast Majority Vote" algorithm due to Boyer and Moore [88]. This algorithm requires only a single pass over the vote buffer. This is accomplished in our DSL using a right fold operation as seen in Figure 22. It is worth pointing out here that it is precisely this aspect, the level of detail in implementing the majority vote, that sets our model of OM(1) apart from previous models.

Finally, we have the system properties declared at the end of `om1` using assertions. Both agreement and validity are predicated on the receivers being done and the values in the list of vote variables.

```
(query
 om1_transition_system
 (let
  ((temp!0 true)
   (temp!1 om1!vote_2)
   (temp!2 om1!vote_1)
   (temp!3 (= temp!1 temp!2))
   (temp!4 om1!vote_0)
   (temp!5 (= temp!1 temp!4))
   (temp!6 (= temp!2 temp!1))
   (temp!7 (= temp!2 temp!4))
   (temp!8 (= temp!4 temp!1))
   (temp!9 (= temp!4 temp!2))
   (temp!10 (and temp!3 temp!5 temp!6 temp!7 temp!8 temp!9))
   (temp!11 (not temp!10))
   (temp!12 om1!recv_2!done)
   (temp!13 om1!recv_1!done)
   (temp!14 om1!recv_0!done)
   (temp!15 (and temp!11 temp!12 temp!13 temp!14))
   (temp!16 (not temp!15)))
  (not temp!15)))
```

Fig. 23 A Sally query rendered in *A*-normal form

3. Generated Model

The formal model generated for OM(1) by LIMA is quite large. Whereas the LIMA specification file is only 5632 bytes, the Sally model LIMA generates for it is 110,581 bytes. The generated model has 65 state variables, 12 input variables, and 16 transitions in total. As an example of what the Sally model looks like, consider the translation of the "agreement" property. Figure 23 shows the corresponding Sally query. The terms in the query are rendered in *A*-normal form [89] to get maximum benefit from sharing sub-terms.

Some verification of these models can be done automatically, without any further work. For example, if we reduce OM(1) system above so that it has only 2 relays and 2 receivers, then the Sally model checker can automatically verify that both agreement and validity hold in just over 11 minutes. With the hybrid fault model assumption made, this is still a fairly non-trivial verification and it is a highly non-trivial one for the model checker to decide.

C. Asynchronous Airbus A320 Autobrake System

1. Informal Description

The third case-study is based on the runway excursion of an Airbus 320, occurring on an Ibiza Airbus on 21st of May, 1998. The full details of and the root cause analysis are detailed in the incident report [86]. The excursion resulted from a total loss of the braking system, due to the simultaneous failure of both channels of the Brake System Control Unit (BSCU) in conjunction with a contamination within the braking system hydraulic system. From the ADSL perspective, it is the failure of the software and BSCU architecture that is of primary interest.

In the A320 system as shown in Figure 24 from [86], the BSCU comprises two channels, with each channel incorporating command and monitor lanes. The command lane provides the active control path to the system, whereas the monitor lane checks and enforces that the command lane is operating within the expected envelope of performance. On the detection of the first failure, the monitoring lane indicates a failure, and control is passed to the other channel. All processing of the system is executed using a quasi-synchronous computational model. That is to say, processing of each channel is quasi-synchronous with respect to one another, and the channels themselves are also quasi-synchronous.

In the Ibiza incident, the root cause of the loss of normal and alternative braking systems, was the Byzantine induced failure of both BSCU channels. The failure was due to the sampling of the auto-brake mode control input panel buttons. The auto-brake panel comprise three buttons, `LO`, `MED` and `MAX` that can be used to select the corresponding auto-braking mode. Momentarily pushing the `LO` button selects the `LO` auto-braking mode. Pressing `LO` again de-selects the auto-braking function, whereas pressing the `MED` or `MAX` buttons selects the corresponding automatic braking

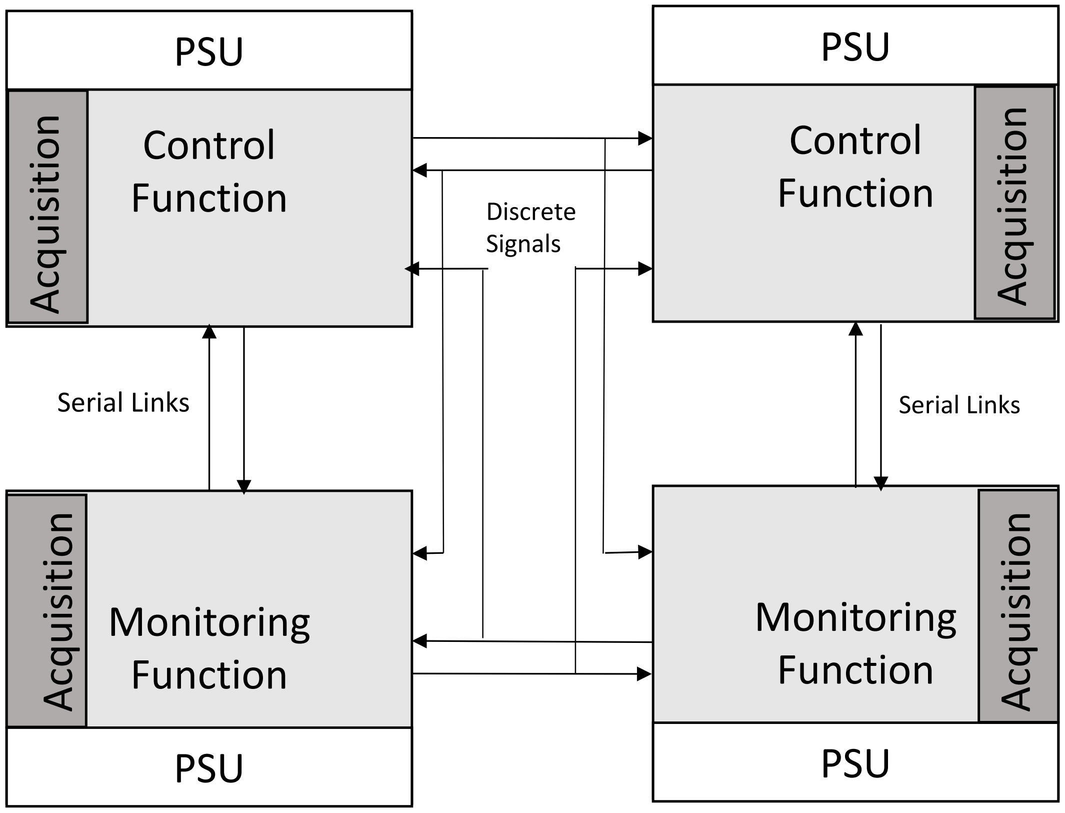


Fig. 24 BSCU Function Schematic from [86].

modes. The state of the selected mode is communicated to the crew by indicators for each mode that are illuminated when the corresponding mode is active.

In the Airbus implementation, the buttons were sampled periodically by software every 25ms. Given the asynchronous composition of the system, each processor samples the button state relative to its own operating time-line. This implementation is vulnerable to short button presses that are too short (< 25ms) and therefore not consistently perceived by all of the sampling units as shown in Figure 25. (Additional inter-lane mode agreement logic, to mitigate Byzantine sampling is not implemented in the system [90].) Hence, the system failure occurred when the command and monitoring lanes of both channels fell into disagreement, following the momentary selection of the `LO` auto-brake mode. The `LO` selection was only detected by one of the lanes of each channel, hence the command and monitor mode disagreement detection logic was erroneously stimulated. In the excursion, the disagreement occurred on both channels simultaneously. Given that the system

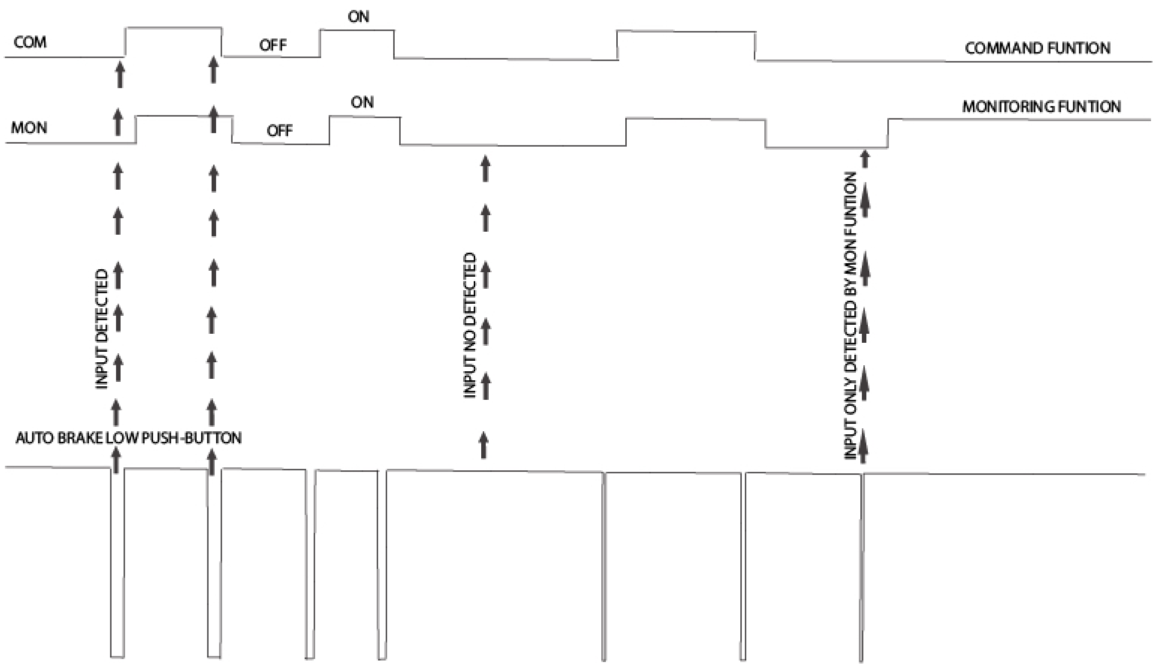


Fig. 25 Oscilloscope chart of pressing times of AUTO/BRK LO vs. acquisition times of COM/MON functions

mode logic was also event driven, there was no path to recovery. Pressing `LO` would be subject to the same Byzantine vulnerability, and even if not Byzantine, the incrementally edge driven mode selection logic would not recover into a consistent state.

The design errors represent a failure uncovered by the traditional design assurance framework. Given the correct-by-construction synthesis focus, it is unlikely that our approach would allow such an implementation to be developed. However, ensuring that the formal synthesis framework is sufficient to represent and explore such design vulnerabilities is a good test case for the ADSL formal tooling. The A320 Airbus brake system offers a good case-study in asynchronous system design and verification. For example, the system also requires a protocol to manage the channel in control, and additional logic and protocols to that communication is healthy. From the incident report these details of the Airbus protocol implementations in these areas are not available. From the description it appears that the channel-in-control logic utilizes an asymmetric power-on timeout to yield a first-up channel in control. This may also leave the system vulnerable to assumptions of BSCU power-on order, and transient recovery strategies. The protocols may also be vulnerable to failures of the inter-lane and inter-channel communication lines. Utilizing the provisions within the LIMA workbench, the long-term goal of this case-study is to support the systematic exploration of candidate protocols and fault models; yielding a better understanding how protocol and communication architecture related design decisions impact core-system properties.

2. Formal Model

The formal model of the Wheel Brake System closely follows the structure of Figure 24. However, to simplify the logic, and to reduce the size of the model, the three buttons of the brake control panel are abstracted into a single button that selects between manual and auto-mode operation. The channel in control is also simplified from the temporal first-up raced based selection, to a fixed-priority scheme, where a pre-configured preferred channel remains in control until it is faulted. Although simpler, this model is sufficient to explore and demonstrate the Byzantine failure vulnerability of the original system.

The formal model starts with a top-level wbs atom that is used to host all the system subcomponents. At the top level, three channels are also implemented, two to convey the button status to each of the lanes, and a third to convey the button state to the lane observer process. The implementation of the lanes leverages and illustrates DSL provisions for parameterized replication. Using a map as shown below, each lane is instantiated with an assigned boolean priority; as described above this priority arbitrates which lane is in control when the system is in full-up operational mode (i.e. no faults present).

```
-- Declare two lanes
 laneIns <- mapM mkLane [True, False]  -- high/low priority
```

The lane implementation comprises two clocked periodic processes, one each for the command and monitor functions, together with an an initialization atom. At every period, the command and monitor sample the input from the button and toggle status of a boolean operational model variable *cautoMode*, on the detection of a rising button edge. At each period, the update status of the *cautoMode* variable is shared with the local lane monitor. The DSL extract for this logic is shown below.

```

   cautoMode <== mux ((value bs ==. Const True) &&.
                      (value prevbs ==. Const False))
                     (not_ (value cautoMode))
                     (value cautoMode)
    writeChannel ctoIn (value framecount)  -- send 'framecount' to
       observer
    writeChannel ctmIn (value cautoMode)
```

The principal periodic process of the monitor atom is symmetrical to the periodic command atom. However, the monitor logic is extended with additional agreement counting logic, to monitor the agreement of the local and command lane exchanged *cautoMode* status. If disagreement persists for three periods, the monitor channel yields control to the other lane, by signaling agreement failure.

```
atom "wait_x_side_autoMode" $ do
      cond $ fullChannel ctmOut
      v <- readChannel ctmOut
      xSideAutoMode <== v
      probeP "monitor.XsideAutoMode" (value xSideAutoMode)

 atom "mon_agreement" $ do
      agreementFailureCount <==
        mux (value mautoMode /=. value xSideAutoMode)
             (Const one + value agreementFailureCount)
             (Const zero)
      -- cond $ value mautoMode /=. value xSideAutoMode
      -- incr agreementFailureCount

```

```
    atom "mon_agreement_count" $ do
      cond $ value agreementFailureCount ==. Const three
      agreementFailure <== Const True
```

The representation of the WBS model in the DSL is very compact with the core logic only requiring about 150 lines of code. When contrasted with the approximately 12,000 lines of Sally code, which the LIMA synthesizes, this is a significant reduction. It may be argued, that without such a DSL and the associated synthesis, the industrial viability of Sally alone may be challenging.

Properties of interest are simply asserted within any of the atoms as illustrated below. However, it should be noted that the variables used within the assert statements need to be within the atom scope. To simply model construction and instrumentation, it is recommended that variables significant to system properties are sent to a top-level observer process, that can provide a central point of property specification.

```
atom "mon_agreement" $ do
  agreementFailureCount <==
       mux (value mautoMode /=. value xSideAutoMode)
           (Const one + value agreementFailureCount)
           (Const zero)
  assert (pName pp "my assert")(value agreementFailureCount <=. Const
      three)
```

In the Airbus braking example, no physical faults were actually present. Hence, in our initial model, we also omit a fault model. However, the DSL framework ensures that the full state of the asynchronous interaction of the sampling and channel in control logic, will be explored within the synthesized Sally model. Therefore, the workbench is anticipated to uncover the system Byzantine failure as part of the formal model analysis.

As part of future work, we intend to augment the fault model of the intra-lane and inter-lane communication channels, and use the DSL and workbench to explore how such failures can impact the assumed system level invariants and safety properties. We also intend to re-introduce the first up leader election protocol, which selects the initial lane in control. One again, we envisage that this will demonstrate how the DSL and formal analysis workbench will support the systematic exploration of how potential faults, and start-up timing variations can disrupt and impact system safety properties and assumptions.

VI. Conclusions

We have laid out a vision in this technical report for an architectural DSL (ADSL) and the associated tools. Our work is heavily driven by real-world case-studies, which we have covered in depth. If anything, we hope to convince the reader that there is an industrial need for simplifying the specification and verification of distributed fault-tolerant systems, and for connecting specifications to their implementations. We have described related work toward this end, as well as our work in building the underlying constructs to support an ADSL. Much of our focus has been on the specification and verification aspects, as we believe that formal proof is necessary for tedious, high-consequence systems.

Although we have promising results and worked case-studies, the future work associated with the research agenda laid out involves significant engineering work we hope to address in coming years. This includes both developing the ADSL workbench itself as well as fleshing out more substantial case-studies.

Acknowledgments

This work is supported by NASA contract #NNL14AA08C. All findings are our own and do not necessarily reflect the opinions of NASA or the United States Government. We particularly thank Wilfredo Torres-Pomales and Paul Miner at NASA Langley for their discussions, insights, and guidance. We thank Tom Hawkins, the original developer of the Atom language that we extend to produce LIMA. Portions of this paper, and particularly Section IV C 1, are adapted from a conference paper [91].

References

[1] D Dvorak et al. Nasa study on flight software complexity. *NASA Office of Chief Engineer*, 2009.

[2] Paul A Judas and Lorraine E Prokop. A historical compilation of software metrics with applicability to NASA's Orion spacecraft flight software sizing. *Innovations in Systems and Software Engineering*, 7(3):161–170, 2011.

[3] Jean-Bernard Itier. A380 integrated modular avionics. In *Proceedings of the ARTIST2 meeting on integrated modular avionics*, volume 1, pages 72–75, 2007.

[4] CM Ananda. General aviation aircraft avionics: Integration & system tests. *Aerospace and Electronic Systems Magazine, IEEE*, 24(5):19–25, 2009.

[5] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, DTIC Document, 2000.

[6] Bruno Dutertre, Arvind Easwaran, Brendan Hall, and Wilfried Steiner. Model-based analysis of timed-triggered ethernet. In *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, pages 9D2–1. IEEE, 2012.

[7] Kevin Driscoll, Brendan Hall, and Srivatsan Varadarajan. Maximizing fault tolerance in a low-s wap data network. In *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, pages 7A2–1. IEEE, 2012.

[8] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (aadl): An introduction. Technical report, DTIC Document, 2006.

[9] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. East-adlâĂŤan architecture description language. In *Architecture Description Languages*, pages 181–195. Springer, 2005.

[10] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. The compass approach: Correctness, modelling and performability of aerospace systems. In *Computer Safety, Reliability, and Security*, pages 173–186. Springer, 2009.

[11] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Access Online via Elsevier, 2011.

[12] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Scheduling and memory requirements analysis with AADL. In *ACM SIGAda Ada Letters*, volume 25, pages 1–10. ACM, 2005.

[13] Yiannis Papadopoulos, Martin Walker, David Parker, Erich Rüde, Rainer Hamann, Andreas Uhlig, Uwe Grätz, and Rune Lien. Engineering failure analysis and design optimisation with hip-hops. *Engineering Failure Analysis*, 18(2):590–608, 2011.

[14] JJ Liu, Shan Zhong, and Hong Ye. Reliability modeling for airborne equipment system using aadl. *Aeronautical Computing Technique*, 39(2):90–94, 2009.

[15] Yue Li, Yi-an Zhu, Chun-yan Ma, and Meng Xu. A method for constructing fault trees from aadl models. In *Autonomic and Trusted Computing*, pages 243–258. Springer, 2011.

[16] Anjali Joshi, Steve Vestal, and Pam Binns. Automatic generation of static fault trees from aadl models. In *Workshop on Architecting Dependable Systems of The 37th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks, Edinburgh, UK*, 2007.

[17] Prototype Verification System (PVS). `http://pvs.csl.sri.com/`.

[18] Dejan Jovanovic and Bruno Dutertre. Property-directed k-induction. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 85–92, 2016.

[19] Systems Modeling System (SysML). `http://www.sysml.org/`.

[20] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.

[21] SAE Architecture Analysis and Design Language (AADL), 2008.

[22] John Rushby. An evidential tool bus. In *In Proceedings of ICFEM 2005*, 2005.

[23] Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. Tool integration with the evidential tool bus. In *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 275–294. Springer, 2013.

[24] RTCA DO-178C: Software Considerations in Airborne Systems and Equipment Certification. `http://www.rtca.org/store_product.asp?prodid=803`.

[25] RTCA DO-254: Design Assurance Guidance for Airborne Electronic Hardware. `http://www.rtca.org/store_product.asp?prodid=752`.

[26] RTCA DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. `http://www.rtca.org/store_product.asp?prodid=617`.

[27] SAE Aerospace Recommended Practice ARP4754, Revision A: Guidelines for Development of Civil Aircraft and Systems. `http://standards.sae.org/arp4754a/`.

[28] SAE Aerospace Recommended Practice ARP476: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. `http://standards.sae.org/arp4761/`.

[29] Pam Binns and Steve Vestal. Formalizing software architectures for embedded systems. In *Embedded Software*, pages 451–468. Springer, 2001.

[30] Pierre Dissaux, Olivier Marc, Stéphane Rubini, Christian Fotsing, Vincent Gaudel, Frank Singhoff, Alain Plantec, Vuong Nguyen-Hong, and Hai-Nam Tran. The smart project: Multi-agent scheduling

simulation of real-time architectures. In *Embedded Real Time Software and Systems*, 2014.

[31] Dominique Blouin, Eric Senn, and Skander Turki. Defining an annex language to the architecture analysis and design language for requirements engineering activities support. In *Model-Driven Requirements Engineering Workshop (MoDRE), 2011*, pages 11–20. IEEE, 2011.

[32] Qi Gong, Duo Su, and Yan Li. Automated safety integration analysis of complex system based on functional model. *Advanced Materials Research*, 655:1783–1786, 2013.

[33] Distributed MILS for dependable information and communication infrastuctures. `http://www.d-mils.org`.

[34] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the bip framework. *IEEE software*, 28(EPFL-ARTICLE-170496):41–48, 2011.

[35] Wenchao Li, Gerard Leonard, and Natarajan Shankar. Design and verification of multi-rate distributed systems. In *Proceedings of the 2015 MEMCODE*. MEMCODE, 2015.

[36] N. Shankar. High-assurance cyber-physical systems. Layered Assurance Workshop (LAW) presentation., 2014. Available at `https://www.acsac.org/2014/workshops/law/Shankar_LAW2014.pdf`.

[37] Paul Pearce and Sanford Friedenthal. A practical approach for modelling submarine subsystem architecture in sysml. In *Proceedings from the 2nd Submarine Institute of Australia (SIA) Submarine Science, Technology and Engineering Conference*, pages 347–360, 2013.

[38] Christiaan JJ Paredis, Yves Bernard, Roger M Burkhart, Hans-Peter Koning, Sanford Friedenthal, Peter Fritzson, Nicolas F Rouquette, and Wladimir Schamai. 5.5. 1 an overview of the sysml-modelica transformation specification. In *INCOSE International Symposium*, volume 20, pages 709–722. Wiley Online Library, 2010.

[39] J Verries and A Sahraoui. Case study on sysml and vhdl-ams for designing and validating systems. In *Proceedings of the World Congress on Engineering and Computer Science*, volume 1, 2013.

[40] Razieh Behjati, Tao Yue, Shiva Nejati, Lionel Briand, and Bran Selic. Extending sysml with aadl concepts for comprehensive system architecture modeling. In *Modelling Foundations and Applications*, pages 236–252. Springer, 2011.

[41] Darren Cofer, Andrew Gacek, Steven Miller, Michael W Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In *NASA Formal Methods*, pages 126–140. Springer, 2012.

[42] Esterel Technologies. Scade. Website, 2015. `http://www.esterel-technologies.com/products/scade-system/`.

[43] Ottmar Beucher. *MATLAB und Simulink (Scientific Computing)*. Pearson Studium, 08 2006.

[44] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[45] B. Scattergood. *The Semantics and Implementation of Machine-readable CSP*. PhD thesis, Queen's College, Oxford, 1998.

[46] Communicating sequential processes for java (JCSP). Website, 2015. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[47] J. Magee and J. Kramer. *Concurrency State Models & Java Programming*. Wiley, 2006.

[48] W. B. Gardner. CSP++: How faithful to CSPm. *Communicating Process Architecture*, 2005.

[49] Thomas Davies. Csp implementation techniques: a critical analysis. Master's thesis, Swansea University, 2012.

[50] Philip Armstrong, Michael Goldsmith, Gavin Lowe, Joel Ouaknine, Hristina Palikareva, A. W. Roscoe, and James Worrell. Recent developments in FDR. In P. Madhusudan and S. A. Seshia, editors, *International Conference on Computer Aided Verification (CAV)*, volume 7358 of *LNCS*, pages 699–704. Springer, 2012.

[51] Symbolic Analysis Laboratory (SAL). `http://sal.csl.sri.com/`.

[52] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *International Conference on Computer Aided Verification (CAV)*, pages 495–499. Springer, 1999.

[53] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[54] John Rushby. SAL tutorial: Analyzing the fault-tolerant algorithm OM(1). Csl technical note, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2004. Available at `http://www.csl.sri.com/users/rushby/abstracts/om1`.

[55] Lee Pike. Model checking for the practical verificationist: a user's perspective on SAL. In *Proceedings of the Automated Formal Methods Workshop (AFM07)*. ACM Press, 2007. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/afm07.html`.

[56] Geoffrey M. Brown and Lee Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2006. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html`.

[57] Lee Pike and Steven D. Johnson. The formal verification of a reintegration protocol. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 286–289, New York,

NY, USA, 2005. ACM Press. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/emsoft.html`.

[58] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, March 2015.

[59] Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 199–214. Springer Berlin Heidelberg, 2004.

[60] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630, 2016.

[61] Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilizing and byzantine-resilient distributed systems. In *CAV*, LNCS, 2016. to appear.

[62] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGOPS Oper. Syst. Rev.*, 40(5), October 2006.

[63] Yanhong A. Liu, Bo Lin, and Scott D. Stoller. Programming and optimizing distributed algorithms: An overview. In *Proc. 8th International Conference & Expo on Emerging Technologies for a Smarter World (CEWIT 2011)*. IEEE Press, November 2011.

[64] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. *High-Level Executable Specifications of Distributed Algorithms*, pages 95–110. Springer Berlin Heidelberg, 2012.

[65] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

[66] Paul Miner, Alfons Geser, Lee Pike, and Jeffery Maddalon. A unified fault-tolerance protocol. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, volume 3253 of *LNCS*, pages 167–182. Springer, 2004.

[67] Lee Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*, pages 231–238. IEEE, 2007. Best Paper Award.

[68] Andrew Gacek, John Backes, Darren Cofer, Konrad Slind, and Mike Whalen. Resolute: an assurance case language for architecture models. In *Proceedings of the 2014 ACM SIGAda annual conference on*

High integrity language technology, pages 19–28. ACM, 2014.

[69] Aeronautical Radio, Inc. Avionics application software standard interface: ARINC specification 653. Online, June 2013.

[70] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[71] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Reliable Distributed Systems Symposium*, pages 93–100, October 1988.

[72] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[73] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. *Byzantine Fault Tolerance, from Theory to Reality*, pages 235–248. Springer Berlin Heidelberg, 2003.

[74] Philip Koopman and Tridib Chakravarty. Cyclic redundancy code (CRC) polynomial selection for embedded networks. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, page 145, 2004.

[75] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

[76] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. Guilt free ivory. In *Haskell Symposium*. ACM, 2015. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/ivory.html`.

[77] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Proceedings of the 1st Intl. Conference on Runtime Verification*, LNCS. Springer, November 2010. Preprint available at `http://www.cs.indiana.edu/~lepike/pub_pages/rv2010.html`.

[78] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14. ACM, 2011.

[79] T. Hawkins. Controlling hybrid vehicles with Haskell. In *Commercial Users of Functional Programming (CUFP)*. ACM, 2008.

[80] Atom documentation can be found at `http://hackage.haskell.org/package/atom`.

[81] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *In Fault Tolerant Computing Symposium 23*, pages 402–411. IEEE Computer Society, 1993.

[82] Bill Potter. Complying with DO-178C and DO-331 using model-based design. 12AEAS-0090. Available at `https://www.mathworks.com/tagteam/74250_Paper%20Number%2012AEAS-0090-finalweb.pdf`.

[83] Lee Pike, Jeffrey Maddalon, Paul Miner, and Alfons Geser. Abstractions for fault-tolerant distributed system verification. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3223 of *LNCS*, pages 257–270. Springer, 2004.

[84] John Rushby. The versatile synchronous observer. In Rohit Gheyi and David Naumann, editors, *Formal Methods: Foundations and Applications*, volume 7498 of *Lecture Notes in Computer Science*, pages 1–1. Springer Berlin Heidelberg, 2012.

[85] Much of the description in this subsection is adapted from a conference paper [].

[86] CIAIAC. Failure of braking, accident occurred on 21 may 1998 to aircraft airbus a-320-212 registration g-ukll at ibiza airport, balearic islands. Technical report, Civil Aviation Accident and Incident Investigation Commission (CIAIAC),, Spain, 2006. Available at `http://www.fss.aero/accident-reports/dvdfiles/ES/1998-05-21-ES.pdf`.

[87] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[88] Robert S. Boyer, Robert S. Boyer, J Strother Moore, and J Strother Moore. Mjrty - a fast majority vote algorithm, 1982.

[89] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, 1992.

[90] There are exceptions; for example, benign faults may be detected by a node itself (e.g., in a built-in-test).

[91] Benjamin F. Jones and Lee Pike. Modular model-checking of a byzantine fault-tolerant protocol. In *Proceedings of the 9th NASA Formal Methods Symposium (NFM)*. Lecture Notes in Computer Science, 2017. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/nfm17.html`.