On the Use of Formal Techniques for Validation*

Neeraj Suri and Purnendu Sinha
Dept. of CIS, NJIT
University Heights, NJ 07102
e-mail: {suri, sinha}@cis.njit.edu

Abstract

The traditional use of formal methods has been for the verification of algorithms or protocols. Given the high cost and limitations in state space coverage provided by conventional validation techniques, we introduce a novel approach to utilize formal verification procedures to drive fault injection based validation of dependable protocols. The paper develops graph structures for representation of information generated through formal processes, as well as a formal framework that facilitates the formulation of specific fault injection experiments for validation.

1 Introduction

As computers for critical applications increasingly depend on dependable and real-time protocols to deliver the specified services, the high, and often unacceptable, costs of incurring operational disruptions becomes a significant consideration. Thus, following the design of protocols, an important objective is to *verify* the correctness of the design and *validate* the correctness of its actual implementation in the desired operational environment, i.e., to establish confidence in the system's actual ability to deliver the desired services. As systems grow more complex with stricter real-time and dependability [9] specifications, the operational state space grows rapidly, and the conventional verification and validation (V&V) techniques face growing limitations, including prohibitive costs and time needed for testing. Thus, the challenges are to (a) identify relevant test cases spanning the large operational state space of the system, (b) do this in a cost-effective manner, i.e., a limited number of specific and realizable tests, and (c) be able to model and validate systems in their entirety (protocol operations, hardware implementations, hardware-software interactions, system load, etc.) instead of the current approaches which stress discrete component validation.

Towards these goals, we investigate and develop techniques to support formal techniques for verification of protocols and develop approaches to utilize verification information to direct the validation of the implementations through the generation of very specific fault-injection experiments. Specifically, our objectives here include:

- To develop a rationale for use of formal techniques towards validation.
- To develop techniques for representation of protocol verification information, and based on these
- To develop a formal framework for guiding and generation of fault-injection (FI) experiments for validation, and present initial experiments to establish the viability of our validation approach.

We emphasize that this is a novel attempt in linking formal methods to validation. Our aim, at present, is to build a basis and perspectives to address these objectives rather than a complete solution.

The organization of the paper is as follows. Section 1.1 provides a background on V&V of dependable operations, and discusses current approaches and their limitations. Section 2 introduces the usage of formal techniques for verification and motivates the proposed validation approach. Section 3 describes the proposed data structures for information representation, and the strategies in organizing the verification information to support validation techniques. We conclude with some current limitations and areas of future research in Section 4.

1.1 V&V of Dependable Protocols : Current Approaches and Limitations

Following design of a protocol, an important aspect is establishing the assurance that the design is fundamentally correct, and that its implementation complies with the requirements to correctly deliver the desired services, i.e., verification and validation.

Currently, verification techniques to establish the correctness of a protocol utilize analytical techniques such as hand proofs, Markovian, Petri Nets, etc. Formal methods [13], a family of mathematical and logical techniques used to reason about computer systems,

*Supported in part by DARPA Grant DABT63-96-C-0044, and NJ-96-421550

are also seeing increasing usage in this verification process. Their main thrust, so far, has been for the verification of algorithms or protocols, and specifically, on finding design stage flaws in algorithms [11, 15, 18].

Validation techniques, typically entail approaches such as modeling, simulations, stress testing, life testing, and also experimental techniques such as fault injection (FI). Given the enormous state space involved in protocols and especially software, analytical, modeling and simulation techniques face coverage limitations. FI based validation is a complex and expensive operation which involves generation of large number of test cases to obtain a reasonable level of confidence in the system operations. Although a wide variety of techniques and tools exist for fault-injection [6], the limitations are the cost, time complexity and actual coverage of the state space to be tested. Two challenges arise: **(a)** how representative are the results to reality? The limitation is in being able to reproduce the actual operational (load, stress, implementation) and failure conditions, and **(b)** how many and exactly which tests need to be conducted? The emphasis here is to scrupulously identify and locate operations which are susceptible to faults.

Statistically, for a critical function with a specified reliability of 10^{-9} failures/hour, 10^{9} hours of fault-free operations need to be tested to expect to uncover even one fault. Not only is the actual state space over this time duration prohibitively large (exercising all possible states is infeasible), but if the failure rate for specific fault types is small, it becomes exceptionally difficult to identify the selected rare fault cases that can cause failures. These problems constitute the fundamental bottleneck of validation and this is where the traditional experimental or probabilistic validation techniques [6] face severe limitations.

Thus, there is need to develop alternate validation techniques, such as the formal methods approach proposed here. As formal methods based on state exploration through techniques such as induction (proof-theoretic approaches) can examine all the behaviors in a very large space of possibilities, thus we investigate the applicability of formal techniques to validation. Overall, our objective is to develop a novel basis for the effective and synergistic use of formal techniques for both verification and validation.

2 Formal Methods Perspectives

Classical fault-injection(FI)[1], though extensively used in establishing confidence in the operation of the fault-tolerance mechanisms of a dependable system, are generally more effective for validation of discrete hardware and software components, i.e., localized fault injection. Our interest extends to validation of general protocols where the operations and capabilities are not only dependent on the underlying resources but also on the implemented resource and redundancy management policies. Two observations (O1, O2) highlight other limitations of the localized classical FI, and thus motivate our research.

O1: Low level (localized) FI may only indirectly influence higher level protocols, thus limiting the scope of FI. Also, faults at protocol level can arise from complex inter linked subsystem events which are difficult to trigger and monitor over the complete protocol. Thus, errors are difficult to trace to fault-observations, especially over protocols.

O2: Furthermore, how realistic and accurate is the state space model for timing and message traffic if the fault distributions are not known or characterizable at the protocol level, either due to low probability of occurrence of fault types (e.g., Byzantine faults), or due to lack of an established fault model, which would preclude the use of existing FI techniques.

Beyond coverage of faults, fault tolerant systems may also be required to deliver system tasks within specified time requirements, i.e., real-time operations. V&V of fault-tolerant protocols is a difficult problem; adding real-time attributes to the protocol further exacerbates the difficulty of verification as well as validation procedures. The main difficulty arises due to the inadequate representation of time and the lack of mechanisms to support the verification process in this aspect. In practice, there are efficient techniques based on discrete-time model which are not severely limited by this problem. However, as a more realistic physical (continuous) time model is used, the continuous time model may require infinite state space as the time component in the states can take arbitrarily real values. There are techniques to construct a quotient space (region graph) out of the infinite space to overcome this problem, though these algorithms tend to be expensive to implement. In an attempt to address such limitations of classical validation, we propose a formal methods directed V&V concept.

Formal methods provide extensive support for automated and exhaustive state explorations over the formal verification to systematically (and formally) analyze the operations of a given protocol. To deal with large state exploration, we choose proof-theoretic for-

[1] An excellent and comprehensive discussion on this topic appears in [6].

mal approaches which utilize logical reasoning, derivations as well as rules of induction to obtain a formal proof basis of the desired system operation. Prior to further discussion, we provide a brief background on formal methods and their capabilities of interest.

Formal methods pertain to mathematical level representation of the system operations. A basic set of assertions characterize the axiomatic basis for the approach – Fig. 1. A "theory" about the protocol properties is encoded as theorems and supporting axioms, and the truth of a theorem is established using rules of inference of the underlying logic used for the specification of the system and its properties. Properties of the protocol are proved by establishing them as logical consequences of the specified axioms, and a proof constructed based on deductive reasoning. This approach provides insights into the specification and its properties such as dependency criteria and boundary conditions. The specific benefits provided by the rigorous application of formal methods include: (a) forces complete, unambiguous and explicit inferences based on the axioms and assumptions, (b) flags conflicting specifications, (c) identifies protocol properties to be validated in reality, (d) specifies requirements on lower-level implementations, (e) supports top-down deductive exploration, and most importantly, (f) supports traceability and reproducibility of actions.

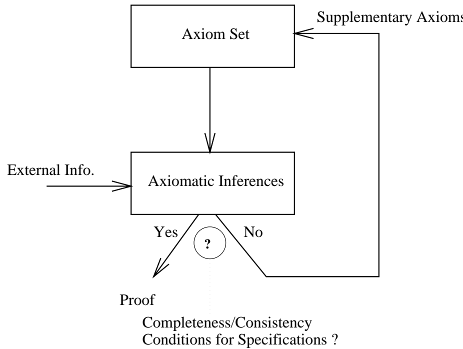

Figure 1: Typical operations of a theorem prover

A variety of formal approaches are currently in use: HOL, EHDM, Boyer-Moore Theorem Prover, PVS, etc.. At the algorithm or protocol level, the need is to be able to support hierarchical operations and hierarchical decomposition of functional blocks. Thus, a high-level logic which can facilitate such a decomposition structure is required. Currently, we use SRI's Prototype Verification System (PVS)[2] tool [12] for our research, although our approaches are applicable to any higher order logic based formal environment.

Overall, our focus is on the issues pertaining to (a) representation of functional and implementational information of the protocol operations, and (b) correlation between specifications and implementations.

3 Formal Techniques for V&V

The accepted objectives of formal techniques are notably different from the requirements of the fault injection process. Thus, our formal methods approach towards V&V and FI based validation of dependable protocols and implementations will involve three specific elements, namely:

1.) Formal specification of protocols with representation/specification of parametric[3] information pertaining to the implementation, and inclusion of these parameters in the formal-method-based verification process.
2.) Representation and visualization of verification information to establish the dependency of operations on specific variables, and to provide mechanisms for modifying parameters, variables and decision operations to enumerate the relevant execution paths of the algorithm.
3.) Identification/creation of suitable fault injection test cases by utilizing visual representation of execution paths, and also propagation paths depicting the scope of influence of parameters and variables on the protocol operations.

We discuss these issues in the following sections.

3.1 Formal Specifications and V&V

The classical use of formal methods has been in formal specification and verification, though very little work exists in incorporating parametric information to the specifications or representations to cover implementation. As our interest lies in developing a *validation* process which essentially requires representation of implementation information, we need to extend the existing specification capability to incorporate parametric information.

To lead into validation, we present examples which **(a)** illustrate the strength of formal methods for verification, and **(b)** highlight aspects that limit applicability of classical formal verification to validation.

Example 1: *FT Clock Synchronization Algorithm*

Consider a distributed system using frame (or "round") based message passing protocols [8, 16],

[2]PVS is being used both for its public domain availability and for its comprehensive theorem proving environment.

[3]E.g., incorporating temporal conditions, replacing clock variables by an actual range of possible crystal frequencies in the implementation, specifying numerical bounds for variables, processor/communication channel bandwidth attributes, implementation features of message passing communication, etc.

where at each frame boundary, each non-faulty node performs the following steps over each successive round.

S1: Each node broadcasts its current personal clock value to all nodes. (*broadcast*)

S2: Each node locally timestamps all received clock values sent to it during that round (within a defined time-stamp interval). (*data assimilation*)

S3: Each node determines a reference value (based on a chosen voting scheme) from the values collected in S2, and computes a correction to align its local clock value to the reference time. (*convergence*)

Additional conditions [15] define the chosen voting strategy, "currently/initially in synchronization" conditions, relative clock skews, specified fault tolerance, time-stamping window size, etc.

This algorithm was used as a case study for formal tools using PVS [15]; a formal verification revealed that the algorithm makes a number of assumptions that are not essential to correct operation. Moreover, it was pointed out in the investigation that a majority of lemmas in the algorithm proof were incorrect although the final proof was correct. The key observation is that formal analysis introduced a higher level of rigor, and identified design (and proof) inconsistencies that were overlooked by both analytical as well as experimental V&V approaches.

However, in validating this algorithm, the implementation involved engineering tweaks that made the validation of the protocol implementation deviate considerably from verification stage, thereby leading to a gap between verification and validation. These tweaks involved defining the operations of the time-stamper, issues of message delivery etc. which we discuss in Section 3.4. The next example further elaborates the need of implementation details in the specification, and being involved in the verification.

Example 2: *FT Real-time Scheduling Algorithm*

In dependable real-time systems, one approach for providing fault-tolerance is by scheduling multiple copies of tasks. Based on a primary-backup approach, a derivative scheme specifies the necessary conditions for tolerating a single fault in the system by establishing conditions on the relative locations of the primary and backup execution intervals. Let r_i, d_i, c_i be task i's release time, deadline and computation time respectively, and $beg(.)$ and $end(.)$ denote beginning and end of task's execution interval. A given condition states that both primary and backup tasks must be scheduled within the task's window[4], and the time interval scheduled for backup should be later than that of the primary, i.e.,:

$$r_i \leq beg(P_i) < end(P_i) \leq beg(B_i) < end(B_i) \leq d_i$$

This condition is required as both primary and backup copies must satisfy the task's timing constraints and because it is assumed that the backup is executed only after a failure in primary is detected. The verification and subsequent validation of the decision procedure can get affected as depicted in the scenario in Fig. 2.

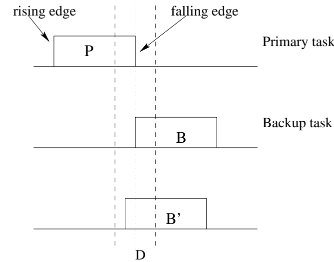

Figure 2: Primary-Backup Execution Intervals

Suppose, the falling edge of primary's scheduled time ends after the rising edge of backup's scheduled time but lies within the chosen granularity, D, in the discrete-time model. Since in the interval-based model of time both events would be considered to have occurred at the same time, this would satisfy the inequality condition, whereas logically it should not. A similar problem arises while scheduling two primary tasks which have dependencies, where one is considered to precede the other. Without a continuous time model, event ordering can always be arbitrarily defined such that any notion of discrete time (regardless of the granularity) can be shown to be inadequate. For validating the implementation of this condition in an actual run-time environment, there is a need to incorporate a continuous model of time in the formal approach.

Simulation-based probabilistic approaches do not necessarily cover all the fault cases due to obvious limitations of not being able to exercise all possible system states in the continuous time domain. However, formal-method-based approaches allow us to conduct speculative experiments as part of the verification process thus investigating a larger design space. For example, we can directly investigate cases where the location of the falling edge of the primary task can be trivially specified to appear either before, at, or after the rising edge of the backup task. Furthermore, these cases of "before" and "after" can be defined for

[4] This is defined as $d_i - r_i$ and is assumed to be twice the computation time.

their duration in time and these speculative cases are then verified for correctness or failure through an iteration of formal verification. The same test process using conventional FI would have required a multitude of test cases covering the entire state space defined as "before" and "after". We have currently formally specified and verified this scheme in PVS using a discrete model of time, and are incorporating a continuous time model to make the validation more realistic by handling concurrent and non-simultaneous tasks.

It is a common misperception to consider formal methods to provide properties of completeness on its own. It *does not* replace informal proof or eliminates testing, but basically, provides for rigor and supplementary aid to proofs and ensures completeness of conditions. Also, even following a correct and rigorous verification, no claims to validation can be asserted until the implementation details are incorporated and reflected in the verification process itself.

3.2 Techniques for Representation of Verification Information to Outline Protocol Execution Paths

On this background, our interest lies in the transformation and the utilization of the information generated by the specification and verification process to aid the identification of system states, and to be able to track the influence path of a variable or implementation parameter to construct a fault injection test case. As stated earlier, the information at the verification stage is in the form of mathematical logic in a syntax appropriate to the chosen formal toolset. However, to aid validation, a fundamental requirement is to visually represent the protocol execution paths generated over the verification process. Another need is to be able to incorporate timing and parameter information at varied levels of abstraction. To this objective, we have developed two novel data structures to encapsulate various information attributes. We label them as **(a) Inference Trees (IT)** or "forward propagation implication graphs", and **(b) Dependency Trees (DT)** or "backward propagation graphs". We present some basic features of these structures prior to discussing their use in validation.

For both IT and DT, we utilize the fact that fault tolerance protocols are usually[5] characterized by forks leading to branches processing specific fault-handling cases [4, 5]. This is a key concept behind validation, which tries to investigate all the possible combinations of branching over time and with parametric information. Both IT and DT are analogous to execution or reachability trees, which elucidate the protocol operations visually. In IT/DT, each *node* represents a function, instruction or decision block of the algorithm, and each *edge* represents the functional, logical, operational and temporal relation between the blocks depicted by the nodes. Each specific source-node to destination-node *path* represents an assertion- and an inference-based activation path of the algorithm. In general, the IT/DT structures share properties with the state transition representations, assertion trees or Petri nets. However, their ability to consider (a) user-defined initiation and termination conditions, (b) conditions for protocol consistency, and (c) no restrictions on the graph acyclicity, distinguishes them from the other approaches. The IT and DT represent graph reachability trees with characteristic capabilities, as discussed in the following sections (3.2.1 and 3.2.2).

[5]This is just a simplification, and not a limitation.

3.2.1 Inference Trees (IT): Forward Propagation Approach

The IT is developed to depict the inference (implication) space involved in a protocol. Each node of the tree represents a primitive *FUNCTION* (or functional block) which is an integral part of the algorithm. Associated with each node is a set of *CONDITIONALS* which dictate the flow of operation to the subsequent *ACTION* as defined in the algorithm. Also associated with each node is the *INFERENCE* space which details the possibility of operations, assertions, and/or usage of event-conditional variables which can be inferred from the node/operation specification. An IT represents the complete set of activation paths of the algorithm (i.e., an enumeration of all operations). Fig. 3 represents the generation of an IT for a majority (2/3) voter. Here, FUNCTION is the 2/3 voter. A set of CONDITIONALS $C[x]$ describes the various conditions (actual or speculative) imposed on the voter. As examples, $x : t-x, t+x$ indicates a message being processed by the voter if it arrives in a specified time window $[t-x, t+x]$, x : *conc* i indicates a message that has to arrive concurrently with message i, and $?i$ queries if all the messages are from the same round i. Based on the inputs to the voter, specific ACTIONS such as the voter outputs, as well as corresponding INFERENCES are generated. An edge between two nodes corresponds to a refinement step incorporating implementation considerations.

The generation of the tree is iterative (see block on top right in Fig. 3). As different conditional (internal or external, parametric, timing) events are desired to be incorporated, a complete verification (and inference) cycle is performed to highlight any inconsistency the new parameters might generate. Implementation

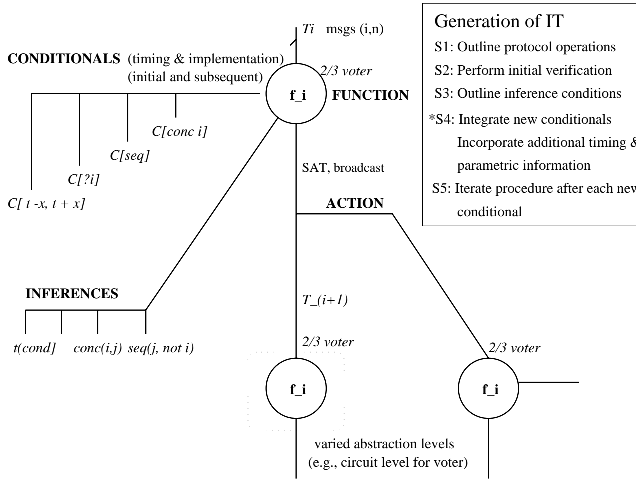

Figure 3: The Inference Tree for a 2/3 Voter Protocol

characteristics[6], action conditionals with concurrency attributes, temporal conditionals, and other similar conditionals get specified at different levels. Basically, each iteration of the IT formulation represents a different level of granularity of system operation. Initially, a high-level IT is constructed with a basic or abstract notion of algorithm operations. As more detailed implementation and operational information are incorporated into the IT, new conditional and associated inference details are generated. We emphasize that each time additional information is modeled into the IT, the verification process needs to be iterated to sustain consistency at all levels of representation. It is of interest to note that the conditional and inference space is dynamically re-generated over each round of verification.

There are no constraints on the graph being acyclic. As we incorporate timing and round information, and as some of the algorithms modeled are iterative in time by nature, path acyclicity is not even desirable. This feature actually allows us to model time and also round based protocol operations. For example, a synchronization algorithm running over multiple rounds can be investigated for properties with messages coming over different rounds by defining a "round number" conditional in the IT.

In the IT, there is no restriction imposed on having specific initiation and termination conditions for any execution path, as is required in the case of assertion trees. IT's facilitate the provision of specifying virtual and temporally established initiation and termination criteria. For example, temporal properties of messages coming over a specific round within a chosen time-frame can be investigated by defining the beginning and end of a given time-frame as initial and termination conditionals in the IT. This feature reduces remarkably the overhead of generating all possible complete[7] execution paths of a protocol. Furthermore, concurrent initiation paths can be established at varying levels of abstraction in the IT. For example, we can set up the same initiation and termination criteria in two different abstraction levels of a function in the IT, one with no timing and implementation information and other one with detailed implementational and operational information. Since these two levels represent different abstractions, the reachability paths from a chosen initiation condition to a specified termination point could be entirely different. This structure provides for mixed levels of abstraction, as a function block can be represented as a complete graph by itself, as for example, in the circuit level abstraction of the voter in Fig. 3.

Currently, we incorporate discrete time variants of classical real-time temporal logics [10][8]. As shown in [1], most timed temporal logics are undecidable in a dense time domain, thus we are investigating (user-interaction-based) approaches to model limited cases (decidable subsets) of dense time.

3.2.2 Dependency Tree (DT): Backwards Propagation Approach

The dependency tree, Fig. 4, is generated by providing in detail information regarding the variables associated with a chosen system operation. These variables are essentially the inference and conditional space provided in the IT. With each protocol operation, we associate a complete list (or speculative list for experiments) of variables which are operated upon during its execution. Deductive logic used by the verifier is applied to determine the actual associated subset of variables. This covers both direct and indirect associations as well as associations over time and rounds among variables. Fig. 4 depicts a multiple round consensus protocol with these characteristics. At each round, the deduction process identifies the variable on which that stage of the operation does or does not depend. For example, in round i, $fn(i)$ does not depend

[6]E.g., processor/channel communication attributes, etc. as relevant to the protocol.

[7]Complete path refers to a path from an initiation condition to a termination point.

[8]Existing timed temporal logics, RTL, MTL, TTL [1, 7], etc., do not easily interface with the inference engine of formal verifiers, though relevant fragments of them can easily be specified and verified in any higher-order logic (say in PVS).

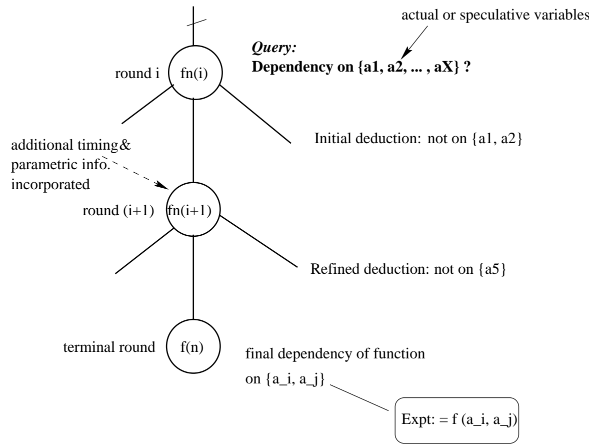

Figure 4: The Dependency Tree : Consensus Example

on variables $a1$ and $a2$. For a distributed synchronization or a consensus operation, identifying and representing the round information is an essential part of the working of the algorithm. Such considerations are very distinctive to a specific algorithm and no attempt is made to classify such considerations through generalized rules. The propagation through the dependency tree is the indicator of the complete set of variables that each facet of the algorithm requires. The leaves of the tree represents the minimal set of dependent variables associated with the primitive function of the protocol. These, in fact, constitute distinct fault injection experiments as the complete propagation path of a system variable with an associated operation is portrayed. Thus, a basic representation of the information gathered over the verification process helps generate a fault injection experiment. The level of granularity of representation of the algorithm and the level of abstraction required for the fault injection process must match for any of these forms of representation to be useful [6].

3.2.3 Representations in IT/DT

Currently, we set up initial IT and DT conditionals based on a thorough understanding of the protocol being tested. This process is iterative across the IT and DT as the initially specified IT conditionals get tested in the DT to ascertain actual protocol dependence on them as conditionals. These conditionals are specified in the PVS theory as axioms, assumptions, numerical ranges and/or numerical constants. Once the specification of the algorithm is complete, we attempt to prove a *putative* theorem which reflects the expected behavior of the algorithm. The success in an attempt to prove the theorem indicates that the set of conditionals chosen earlier are sufficient enough to satisfy all the assertions made in the specification. A failure in the proof process indicates that either the conditions specified are not sufficient or the proof strategies are not correct, or even that the statement of the query is not phrased properly. Failures also reveal conditions which were not being satisfied. A successfully completed verification process also provides a list of functional dependencies on various assumptions. Based on these inferences, a new set of conditionals is added or an existing set is modified. This feature provides us with the capability of speculatively pose new or change conditionals to observe the behavior of the system. We still need to generate an automated process for defining the relevant conditionals. As an initial approach, we are investigating the possibilities of first automating the cases in the DT and then using the generated function dependencies to specify the conditionals for the IT. In this respect, we are developing mechanisms for describing and providing feedback across IT/DT.

Based on IT and DT interactions, we compute the INFERENCE space knowing the CONDITIONAL and ACTION spaces. For example, consider Fig. 3: we can specify a condition $C[conc\ \ i]$ in the CONDITIONAL space and pose query "Is message j concurrent with i?" in the DT. It may then ask us to specify a time-window within which the two messages are to be considered, in which case, we need to add an extra conditional specifying a time-window and re-run the query, or it may simply confirm that message j has arrived concurrently with message i, which gets reflected as an inference $conc(i, j)$ in the INFERENCE space. We are also looking into analyzing the nature of and the depth of information provided in INFERENCE and CONDITIONAL spaces.

We have incorporated a basic capability for adding parametric information which allows us to cover different levels of system representation as well. As each iteration of the IT formulation represents a different level of granularity of the system operations, we are looking at issues, such as degree of details to be incorporated, related to the interaction of inferences and queries at different levels of abstractions. The key observation is that faults to be injected are basically derived by queries related to the potential discrepancies between the levels. Currently, we can specify and interface the specifications of the 2/3 voter at both the protocol and at the circuit level in PVS. For a more detailed specification, we are looking at VHDL or BDD level descriptions of gates/devices. We are also looking at defining interfaces to link the VHDL

and BDD level specifications to the PVS syntax and inference engine.

3.3 Validation: Defining the Fault Injection Test Cases

The advantage of our approach is that the set of fault injection tests generated will be comprehensive to the extent of implementation details modeled into the formal specification, i.e., protocol, circuit, gate level, etc. As the example in Section 3.4 demonstrates, the DT results can pinpoint a specific block to be modeled to a refined level of detail as needed. Each reachability path in the tree provides for a test case. As the verification process is re-executed over each introduction of conditionals or parameters, it eliminates the possibilities of new faults being introduced. A simple observation is that the cases generated through this process involve *all* relevant test cases; rare test cases being necessarily included.

The key element here is to sustain, at all time, the axiomatic rules under which the protocol verification stays valid. This suggests a situation that the set of conditionals are not fixed on *a priori* basis. Each round of iteration can generate constraining conditions which in turn get reflected as new conditionals. There is a possibility of a deadlock condition getting generated simply as a result of the iteration process. However, such a generated condition is a direct reflection of an erroneous operational condition. Actually, it is simpler to utilize the dependency graph in this situation as such a condition will actually be automatically flagged by the verification process.

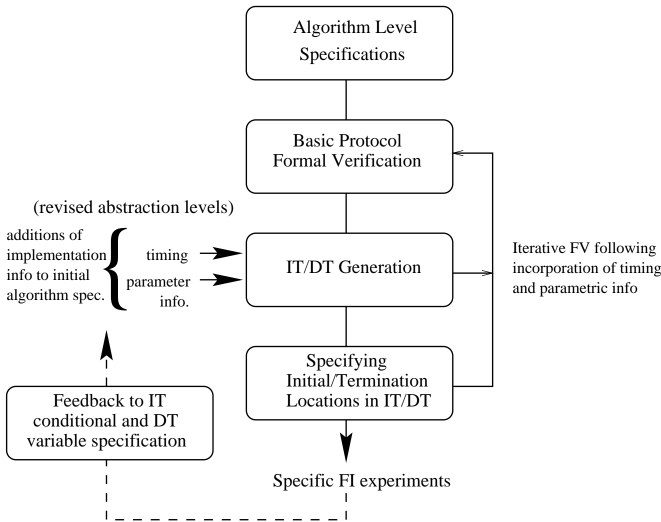

Figure 5: Generating the FI experiments

Fig. 5 represents the general process of generating FI experiments using IT's and DT's. Below, we highlight specific aspects of IT's and DT's in generating FI experiments:

- As each reachability path in the IT potentially defines a FI experiment, we have the flexibility of choosing a single path, or having multiple initiation instances merging into a single termination point. An experimental setup can be across different levels of abstraction (i.e., a message over a channel is modeled as a bit stream inside a voter which in turn is modeled at the circuit level), and also over different time instances (a synchronization protocol has variables which have effect over multiple rounds of synchronization).
- The DT provides flexibility for conducting exhaustive checking. At each iteration, the dependency list is pruned as one progresses along a reachability path. At any desired level, the elements of the current dependency list constitute the variables to be tested, i.e., the FI experiment.
- Path activations and terminations in either IT or DT can be specified by associating counters and timeouts. Thus, transient fault cases are incorporated by (a) specifying a start condition for the transient, and (b) removing the condition after a desired interval over any chosen path/branch in the IT/DT. This approach facilitates us in defining multiple paths, concurrent events, as well as paths reflecting either the complete or partial protocol operation.

Overall, this approach generates a pseudo-simulation environment, except that this is completely deterministic and reproducible. Thus, we not only have a capability of performing basic validation, but also a design tool to perform speculative changes at the protocol and implementation level and observe the impact. It also provides a direct capability of tracing the propagation path of any variable (or fault) via reachability analysis. As the IT/DT needs only a reachability path to define an experiment, we can also generate fault injection cases over any desired feasible path without an overall termination condition for the function.

3.4 Initial V&V Results: Clock Synchronization Example

These proposed validation techniques were tested on an actual implementation of the clock synchronization algorithm [8, 17] presented in Section 3.1 (Step S2), where the incoming clock signals at the recipient nodes are time-stamped (based on the recipient's clock value) in the order they were received – Fig. 6

where A is the recipient node and messages from B, C and D are time-stamped based on A's local time. This ensures that the temporal ordering of messages is maintained.

According to the implementation requirements, the time-stamper unit has multiple input channels but as it processes only one channel at any given moment, the messages get automatically sorted. There is a simplistic underlying assumption that there will be a certain distance in time between the signals and thus no concurrent timestamps related conflicts will arise.

Over the verification process, all assumptions for synchronization were maintained and the protocol was considered verified. In the implementation, the time stamper was provided with a specification regarding the distance in time between two successive incoming messages for the purpose of serializing them in time. However, this specification was inadequately (and incorrectly) specified and implemented such that this distance was actually longer than the time distance between two perfectly synchronized clock messages. Thus, if two clock messages came synchronized and closer in time than the specified time distance in the time stamper, it would default and adopt a random polling mechanism, and thereby creating a partial ordering problem (Fig. 6 right side).

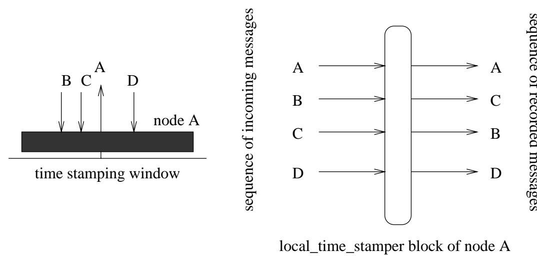

Figure 6: Clock Synchronization - Timestamper

This implementation had been extensively tested using classical fault injection techniques [17] and this condition was not discovered. However, subsequent to the synchronizer block, a consensus block recording the actual time-ordered set of nodes in the synchronized set, would show variations in the message sequences (as in Fig. 6) thus indicating some deviation. Using conventional FI approach based on both accelerated testing and random fault injections, 2.7 million test cases were injected with 5 faults detected, but the discussed fault case was missed. The IT/DT approach generated a total of 310 experiments and identified 9 faults (the discussed case and 8 others including the 5 faults found in the classical approach) including the partial ordering case. It is unusual that the erroneous time stamping situation appears when the system is working perfectly with the clock signals arriving very close to each other. This case disappears when the system has a high load or in the case of a fault where the incoming messages get staggered further in time than the specified time distance in the time stamper, thus meeting all specified requirements.

The IT and DT of the synchronizer block were set up, similar to Fig. 3 & 4, to model the synchronization protocol. In this block, no errors in either the protocol or the implementation were found. However, the DT of the subsequent consensus block declared order dependency on the convergence block. Next, the DT in the convergence block determined ordering function dependency on the timestamper block. Remodeling the IT of timestamper at the VHDL level highlighted the implementation problem. We make three observations here: (a) the fault propagation extended over different functional blocks of the overall protocol, (b) iterative use of the IT/DT over different blocks helped identify the exact function/block, and (c) the timestamper block needed modeling to a more detailed VHDL level based on this block's specific identification over the DT processes.

The 310 test cases generated using the IT/DT approach provide for validation which is exhaustive only to the number of specified parametric inputs such as the functional description of the time stamping unit. Fortunately, this amount of parametric information sufficed to pinpoint the fault case[9]. In a general setting, the number of test cases could have been higher had the specification required more information to ascertain the exact dependency of the synchronization algorithm on the timestamper. However, as we are selectively and iteratively determining the dependency of a given functional unit on the input parameters, the number of tests required is significantly less than that for random or statistical testing.

A similar test was conducted on the 2/3 majority voter. In this case 3827 tests were needed using classical FI versus 24 tests identified by the proposed formal methods assisted techniques. In both cases, the implementation had 3 fault cases and both techniques were correctly able to identify them.

Fault injection, in general, is a probabilistic validation approach, and our formal approach does not make any claim of completeness of validation. However, with exhaustive state exploration possible via formal techniques, we do expect to develop capabilities of reaching "closer" to a complete validation scenario,

[9]Determining, *a priori*, the level of detail needed to be represented is an open problem.

once an automated form for generation and testing over IT/DT's can be accomplished.

These are early results that we present in this paper to highlight the effectiveness of the proposed approach. We acknowledge that we need to cover a variety of classes of protocols before claiming the overall effectiveness of our approach.

4 Conclusions, Limitations and Future Directions

The current V&V techniques are limited in handling the large state space involved in high dependability operations. We have introduced a new approach to FI based validation which extends the domain of formal techniques beyond verification to generate novel validation strategies for dependable operations.

Currently, we have introduced techniques for the representation of information generated over the specification and verification process. We have developed the basic guidelines for generating IT and DT, and are developing detailed approaches to the incorporation of dense/continuous time considerations. These will allow us to conduct V&V of real-time protocols which are currently very difficult to test using classical FI techniques. We have yet to fully incorporate the specification of system load (and stress) into the formal engine. At present we are limited to approximating these conditions using distributions; in the future we are looking at approaches to model stress and load as parametric inputs. We are also currently investigating approaches to formally specify (and interface) various levels of abstractions over the implementation stages. For example, the ability to formally model at the block level, systematically leading to a specification at the circuit, gate and device level is a significant challenge that we plan to address.

A current limitation is the need of a specialized PVS syntax to perform the formal specifications or to pose the deductive queries in the DT. Our intent in the future is to develop a GUI interface to simplify this step. Given the features of our proposed approach, we envision our techniques to complement conventional FI techniques to provide for improved protocol validation. To this extent, we are looking at automating and interfacing the IT/DT generation and iteration process to other existing FI toolsets such as DEPEND.

As mentioned in the abstract, we have introduced initial approaches to validation using formal techniques. A few simple examples have been presented to show the viability of this approach. Subsequently, we plan to refine the approach to make it amenable for practical V&V of dependable operations.

References

[1] Alur, R., Henzinger, T. A., "Logics and Models of Real Time: A Survey." *Real Time: Theory in Practice*, (J.W. de Bakker, K. Huizing, W. de Roover, G. Rozenberg, eds.), LNCS 600, Springer-Verlag, pp. 74–106, 1992.

[2] Arlat, J. *et al.*, "Fault Injection for Dependability Validation," *IEEE Trans. Software Engineering*, vol. 16, pp. 166–182, Feb. 1990.

[3] Avresky, D. *et al.*, "Fault Injection for the Formal Testing of Fault Tolerance," *FTCS-22*, pp. 345–354, 1992.

[4] Echtle, K., Chen, Y., "Evaluation of Deterministic Fault Injection for Fault-tolerant Protocol Testing," *FTCS-21*, pp. 418–425, 1991.

[5] Echtle, K. et al, "Test of Fault Tolerant Systems by Fault Injection," *FTPDS, IEEE Press*, pp. 244–251, 1995.

[6] Iyer, R. and Tang, D., "Experimental Analysis of Computer System Dependability," *Book chapter in 'Fault Tolerant Computer System Design'*, editor: D.K. Pradhan, Prentice Hall, pp. 282–392, 1996.

[7] Jahanian, F. and Mok, A., "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. on Software Engineering*, pp. 890–904, Sept. 1986.

[8] Lamport, L. and Melliar-Smith, P. M., "Synchronizing Clocks in the Presence of Faults." *JACM*, 32(1), pp. 52–78, Jan. 1985.

[9] J-C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology." *Proceedings of FTCS-15*, pp. 2–11, 1985.

[10] Manna, Z. and Pnueli, A., "Verification of Concurrent Programs: The Temporal Framework," *TR STAN-CS-81-836*, Stanford Univ, 1981.

[11] Owre, S. *et al.*, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Trans. Software Engineering*, Jan. 1995.

[12] Owre, S., Shankar, N., *The Formal Semantic of PVS*, SRI-CSL-97-2, Aug. 1997.

[13] Rushby, J., "Formal Methods and the Certification of Critical Systems," *SRI-TR CSL-93-7*, Dec. 1993.

[14] Rushby, J., "A Formally Verified Algorithm for Clock Synchronization Under a Hybrid Fault Model." In *ACM PODC*, 304–313, 1994.

[15] Rushby, J. and von Henke, F., "Formal Verification of Algorithms for Critical Systems." *IEEE Trans. Software Engineering*, vol. 19, pp. 13–23, Jan. 1993.

[16] Suri, N., Walter, C. and Hugue, M., "Synchronization Issues in Real-Time Systems," *Proc. of IEEE,* vol. 82, no. 1, Jan. 1994.

[17] Walter, C., "Evaluation and Design of an Ultra-Reliable Distributed Architecture for Fault Tolerance," *IEEE Trans. on Reliability*, Oct. 1990.

[18] Walter, C., Lincoln, P., and Suri, N., "Formally Verified On-Line Diagnosis," *IEEE Trans. on Software Engineering*, vol. 23, no. 11, Nov. 1997.