A Pre-Injection Analysis for Identifying Fault-Injection Tests for Protocol Validation

Neeraj Suri
TU Darmstadt, Germany
Email: *suri@cs.tu-darmstadt.de*

Purnendu Sinha
GM R&D, Bangalore, India
Email: *purnendu.sinha@gm.com*

***Abstract*— Fault-injection (FI) based techniques for dependability assessment of distributed protocols face certain limitations in providing state-space coverage and also incur high operational cost. This is mainly due to lack of complete knowledge of fault-distribution at the protocol level which in turn limits the use of statistical approaches in deriving and estimating the number of test cases to inject. In practice, formal techniques have effectively being used in proving the correctness of dependable distributed protocols, and these techniques traditionally have not been directly associated with experimental validation techniques such as FI-based testing. There exists a gap between these two well-established approaches, viz. formal verification and FI-based validation techniques. If there exists an approach which utilizing a rich set of information pertaining to the protocol operation generated through formal verification process can provide guided-support to perform FI-based validation, then the overall effectiveness of such validation techniques can be greatly improved. With this viewpoint, in this paper, we propose a methodology which utilizes the theorem-proving technique as an underlying formal-engine, and is composed of two novel structured and graphical representation schemes (interactive user-interfaces) for (a) capturing/visualizing information generated over the formal verification process, (b) facilitating interactive analysis through the chosen formal-engine (any theorem-proving tool) and database, and (c) user-guided identification of influential parameters, those eventually used for generating test cases for FI-based testing. A case study of an on-line diagnosis protocol is used to illustrate and establish the viability of the proposed methodology.**

***Index Terms*— Dependable Distributed Protocols, Fault Injection, Formal Techniques, Verification and Validation.**

I. INTRODUCTION

Computers for critical applications increasingly rely on dependable protocols to deliver the specified services. Consequently, the high (and often unacceptable) costs of incurring operational disruptions become a significant consideration. Thus, following the design of dependable 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 composite real-time and dependability [32] 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. Fault injection (FI) techniques have commonly been used in practice for validating system's dependability. Although a wide variety of techniques and tools exist for FI [30], the limitations are the cost, time complexity and actual coverage of the state space to be tested. In these respects, the challenges are to (a) identify relevant test cases spanning the large operational state space of the system, and (b) do this in a cost-effective manner, i.e., a limited number of specific and realizable tests. It has been analytically shown in [19] that deterministic fault injection provides benefits over random fault injection in protocol testing. In this context, a pre-injection analysis that aims at identifying a key set of variables/parameters of the given dependable protocol which would constitute test cases for FI experiments can strongly help to minimize/reduce the number of test cases.

Typical examples of protocols widely used in dependable distributed systems include: clock synchronization, consensus, checkpointing & recovery, and diagnosis, etc. [38], [48]. For V&V purposes, algorithmic description of these dependable distributed protocols can be specified using a formal specification language that supports high-level modeling constructs including hierarchical decomposition, recursion, parameterized functions, etc. With proof-of-correctness of the algorithm established using inference-rules of the chosen logic, we aim at exploiting this verification information to

doi:10.4304/jsw.5.10.1144-1161

support and supplement FI-based validation of dependable distributed protocols. Our objective is to systematically determine fault-cases by looking into various assumptions which influence the protocol operation and also inter-dependencies among different system components. This particular aspect forms the basis for our proposed pre-injection analysis. The novel contribution of our proposed techniques is in developing usable links across formal verification and experimental validation approaches. Specifically, to demonstrate the viability of our proposed research in formal-method-guided pre-injection analysis, we have:

- Developed two novel representation schemes (Inference Tree (IT) and Dependency Tree (DT)) to visualize protocol verification information and facilitate interactions with the underlying formal engine and database for analysis.
- Based on the IT/DT, (a) outlined the deductive capabilities of our formal-method-based query processing mechanisms, and (b) developed a methodology to select and identify parameters which would constitute test cases for FI experiments for validation.
- Discussed a tool implementation which generates test cases for FI experiments, i.e., formally driven pre-injection analysis.
- Demonstrated the practical effectiveness of formal techniques for guiding classical FI experimentation through identification of pertinent test cases for validating an online diagnosis protocol.

Organization: Section II presents an overview of FI-based dependability validation as well as a short note on formal methods highlighting key aspects of formal modeling of distributed protocols. Our proposed approach for pre-injection analysis is described in Section III. Section IV presents a case study of a dependable distributed protocol, namely online diagnosis protocol demonstrating the effectiveness of our proposed pre-injection analysis for identifying test cases to guide FI-based protocol testing. Section V provides a comparative view with other related work. We conclude with discussions in Section VI.

II. Background

In this section, we first provide a background on fault-injection based dependability validation and then give an introduction to formal methods.

A. An Overview of Fault-Injection based Dependability Validation

Validation techniques typically entail approaches such as modeling, simulating, stress testing, life testing, and fault-injection (FI) [30, Chapter 5] based testing. FI involves the process of deliberately injecting faults (into the actual system or system model/simulation) to test the effectiveness of the dependability mechanisms designed to contain the errors resulting from the injected fault. From the perspective of experimental validation, classical FI is extensively used in establishing confidence in the operation of the fault-tolerance mechanisms of a dependable system. FI based validation is very effective provided (a) accurate and detailed representation of the system and its operations is available, and (b) the selection of FI experiments is appropriate to stimulate the system to ascertain a desired level of testing confidence. It has been shown in [30] that usually an extremely large number of faults need to be injected in order to obtain a small interval estimate at a high confidence level, particularly if the desired coverage value is very high. Thus, from a realistic viewpoint, a basic issue in FI-based approaches is the selection of specific (ideally, a minimum number of) test cases to inject as it is not possible to carry out an extremely large number of fault injection within practical time/cost constraints.

For specific systems where the nature of the workload (e.g., real applications, selected benchmarks or synthetic programs), nature of fault distribution and operation domain is well defined, the random FI techniques work quite effectively [30], [57]. The realism and accuracy of the state space model for timing and message traffic degrades rapidly if the fault distributions are not known or characterizable at the protocol level. This is either due to low probability of occurrence of rare but significant fault types (e.g., Byzantine faults), or due to lack of an established fault model. In such cases, the premise of random FI breaks down as the statistical basis of selecting random test cases is no longer valid. This aspect thus precludes the use of existing FI techniques that use distributions to derive maximum likelihood estimates to determine the number of test cases for a desired confidence interval.

B. A Short Introduction to Formal Methods

Formal methods provide extensive support for automated and exhaustive state explorations over the formal verification to systematically analyze the operations of a given protocol. To deal with large (potentially infinite) state exploration, we choose proof-theoretic formal approaches which utilize logical reasoning, derivations as well as rules of induction to obtain a formal proof basis of the desired system operation. The primary reason for using theorem proving approaches is that a proof-tree can be obtained and associated proof-analysis can facilitate identification of relevant set of variables. We refer the reader to [46, Section 2.2] for a detailed comparison of proof- and model-theoretic approaches.

Formal Methods for Distributed Actions

Distributed protocols can be seen, from a modeling point of view, as sequences of deterministic operations

interleaved with branching points, where the *Function* (or algorithm) takes decisions based on the actual information it has obtained. We can call such sequences of deterministic operations as *Actions*. In a proof-theoretic context[1] we can prove the fact that an action implements the specified behavior as a theorem. That is, for each action we can try to build a proof that, starting from some given axioms or *Conditionals* certain *Inferences* can be drawn out, which correspond to the possibility of operations, assertions, and/or usage of event conditional variables. Each action, being deterministically defined, can be modeled as a set of predicates. Using these predicates, we can try to prove certain conjectures (i.e., unproven theorem) starting form the conditions given as hypothesis. Using the resulting inferences, it is possible to determine: (a) which alternative branch will be chosen after an action completes; (b) which are the conditions for the next action; (c) whether the protocol implements the specified and desired properties.

PVS Tool Support

At the 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. For our studies, we used SRI's Prototype Verification System (PVS) tool [40] for our research, although our approaches are applicable to any higher order logic based formal environment. PVS provides a powerful interactive proof-checker with the ability to store and replay proofs. The PVS system provides several commands for determining the status of theories, such as whether a proof has been performed/completed. Proof-chain analysis, an important form of status report, assures that all the proof obligations are fulfilled. It also identifies the axiomatic foundation of the given theorem, i.e., it analyzes a given proof to determine its dependencies.

III. Proposed Pre-Injection Analysis

Formal methods have primarily been used as verification techniques (i.e., to capture conformance to design specification) in establishing correctness of the design. On the other hand, experimental testing targets actual implementations. Obviously a gap exists to transcend from abstract properties to implementation details. This research aims at bridging the gap between formal verification and experimental validation/testing. Towards this aim, our key contributions include development of:

- A methodology for pre-injection analysis which involves techniques for representation and visualization of verification information to establish the dependency of operations on specific variables as represented in formal specification of the protocol. Moreover, the developed techniques provide mechanisms for modifying parameters, variables and decision operations to enumerate the relevant execution paths of the protocol. This is achieved by updating the formal specification of the protocol and verifying the properties of interests through the underlying formal-tool .
- An approach for identification/creation of suitable and specific FI test cases. It is achieved by utilizing representation of execution paths as well as propagation paths depicting the scope of influence of parameters and variables on the protocol operations.

Before describing the proposed methodology for formal-methods driven FI-based validation process, it is necessary to briefly introduce the two key structured verification-information representation schemes.

A. Representation and Visualization of Verification Information

Typically, after developing the formal specification of a protocol and its subsequent formal verification, the information at the verification stage is in the form of mathematical logic in a syntax appropriate to the chosen formal tool-set. As our interest is in protocol validation, we need to transform and utilize 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 FI test case. Towards this objective, we have developed two structured representation and visualization schemes to encapsulate various information attributes. We label them as (a) *Inference Tree (IT)* or "forward propagation implication tree", and (b) *Dependency Tree (DT)* or "backward propagation deductive tree". An IT outlines the inference conditions and the actions taken during the verification process, while a DT captures the variable/functional block that the protocol/specification rely on. Moreover, DT facilitates query processing and/or 'what-if' analysis on the information accumulated over the verification process. We present some basic features of these structures prior to discussing their complementary use in validation.

We observe that most dependable protocols consist of decision stages leading to branches processing specific error-handling cases [5], [10], [19], [20], [54]. This is a key concept behind validation, which tries to investigate all the possible combinations of branching over time and with parametric information (examples include numeric bounds for variables, round number, processor attributes,

[1] An axiomatic theory consists of a number of primitive terms and set of statements which are true within that theory (known as axioms). A proof in a theory is a finite sequence $S_1, S_2, S_3, \ldots, S_n$ of statements in the theory such that each S is an axiom, or can be derived from any of the preceding statements by applying a rule of inference (such statements are known as theorems).

communication bandwidth, etc.). The proposed IT structure elucidates the protocol operations visually, and has the capabilities to capture various subtleties (set of variables/ event-conditionals, inferences, etc.) being generated over each round for round-based protocols obtained via formally verifying the protocol specification. The complementary structure DT establishes the dependency of the protocol operations on these variables /conditions. The set of variables appearing in the dependency list is essentially used in formulating the FI experiments.

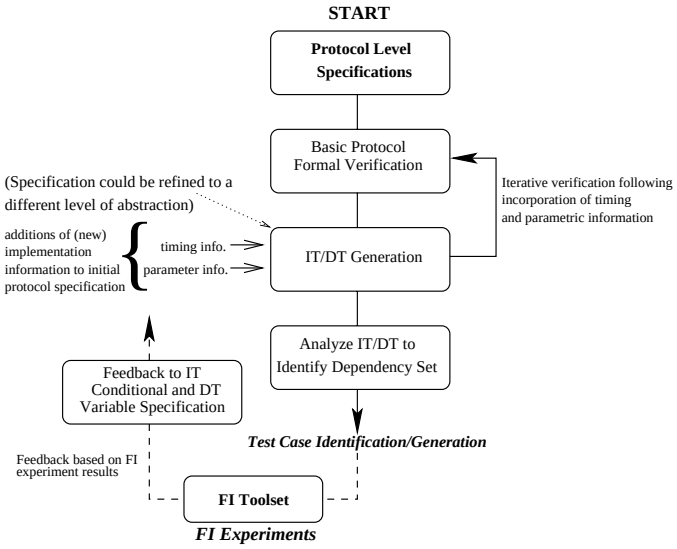

Fig. 1. Overall Process of Generating FI Experiments

B. Proposed Methodology for FI-based Validation

Fig. 1 depicts the overall process of FI experimentation using the IT and DT approach. We emphasize that our pre-injection analysis is *iterative* in nature primarily to work with different levels of abstraction as well as to facilitate speculative or "what-if" type of analysis. The following steps are utilized to aid the FI process:

Step 1: Formally specify the protocol operations and desired properties of interests.

Step 2: Perform initial formal verification to demonstrate that the specification conforms to the system requirements.

Step 3: Generate the IT/DT utilizing the verification information and generated inferences to enumerate the execution paths and establish the dependency of the operations on design variables through DT.

If any new information pertaining to specific implementation-level details (e.g., list of variables/event-conditionals) is added in the formal specification of the protocol, the specification needs to be verified to flag any inconsistencies.

Step 4: Analyze IT/DT to identify deductively dependencies of these variables/conditionals and based on this information select parameters and/or functional blocks to generate test cases for FI.

The resulting test cases form the basis for FI experiments. It is to note that the output of (or observations from) the FI experiments could also trigger addition/deletion of certain constraints on variables or implementation-specific details about the variables in the formal specification. This then needs to be followed up with the iterative verification process to sustain consistency at all levels of representation

Step 5: Design FI experiments from these test cases based on the chosen FI tool-set (e.g. [27]).

Note that our main intent is pre-injection analysis in identifying the test cases. For completeness, fault-injection related steps have been mentioned. Feedback obtained over the actual FI experiment can be fed back to the IT/DT process. Observations from FI experiments could also guide addition/deletion of implementation-specific information in the formal specification of the protocol.

Inference Trees (IT): Visualizing Protocol Execution

IT outlines the governing conditions, inferences and the actions taken during the verification process. This representation structure is developed to depict these key aspects over the execution of a protocol. We next describe the process to generate the IT, that is, **Step 3** mentioned above. Recall that successful completion of formal verification through the underlying formal engine is a pre-requisite for generation of IT.

Step A: Based on the verification process, for a particular round of protocol operation and a specific functional block, outline governing conditions, resulting inferences and an action taken or an alternative action to be taken.

Step A.1: Repeat the same for subsequent rounds of the protocol operation based on the verification process. Stop after the final round of the operation.

Step A.2: If no new information to be added/incorporated, Stop.

Step B: For speculative "what-if" analysis, interactively add new conditionals in terms of new timing, parametric or operation information in the specification language of the underlying formal engine and perform formal verification of the modified formal specification of the protocol.

Step B.1: Based on the verification process, update the resultant inferences, newly added conditionals and actions taken.

Step B.2: If no new information to be added/incorporated, Stop.

Step C: Iterate **Step B** for each new condition being introduced.

We first present a generic description of the IT and then follow up with detailing different aspects of it through a specific case study. Please refer to Figure 2 to relate the terms described next. Each node of the tree represents a primitive FUNCTION (or a functional block/ an algorithmic step of the protocol) at a given level of abstraction. Associated with each node is a set of CONDITIONALS (assumptions specified as axioms in the formal specification) which dictate the flow of operation to the subsequent ACTION(s) as defined for the protocol. Also associated with each node is the INFERENCES space which details the possibility of operation (or sequence of operations), assertions, and/or usage of event-conditional variables which can be inferred from the node/operation specification. A particular inference could potentially update the conditionals for the subsequent round of protocol execution where a specific action will be taken. Note that FUNCTION, CONDITIONALS, INFERENCES and ACTION are constituent part of the IT structure. Furthermore, a connection (edge) between two nodes/functional blocks represents a logical or temporal relation in terms of algorithmic actions/steps taken based on the prevailing conditions. A path between two nodes comprising of multiple connections represents a set of actions taken up by the protocols.

The set of CONDITIONALS consists of two parts: (i) the basic algorithm (definitions), assumptions, and constraints, and (ii) postulated properties (claims) about the protocol. Thus, initially, the CONDITIONAL space contains only the basic assumptions and constraints for the given protocol, and basic derivative properties. Over subsequent verification rounds, the CONDITIONAL space is enhanced with more information about parameters that may impact the behavior of the protocol. Note that both CONDITIONALS and INFERENCES are formally obtained from the protocol specifications. In fact the theorem prover process defines the conditionals as requisite stopping conditions to be satisfied prior to proceeding to a subsequent step in a proof. Using functional level specification of the protocol, an IT represents the complete set of activation paths of the protocol (i.e., enumeration of all operations). It is important to point out that the process of generating CONDITIONAL and INFERENCE spaces are semi-automatic and involves users intuitions and understanding of formal specification, and the implications of the proofs. Moreover, it is notable that both CONDITIONAL and INFERENCE spaces can grow or shrink depending on the protocol and its operating conditions, though the growth of these two spaces are linearly bounded by the system parameters.

In order to keep track of influences of newly added conditionals on the protocol operation, the IT structure facilitates recording of inference(s) leading to specific action(s) (we label them as "leads to this action") as well as resulting inference(s) updating the conditionals for the subsequent round of protocol execution (we label them as "updates ... operation").

Another key feature of the IT is that it provides for mixed levels of abstraction, as a function block can be represented as a complete graph by itself. For example, the voter function can be represented at the circuit level abstraction and modeled in say RTL-level specification as shown in Fig. 2 (the lower right-most node).

An Illustration of the Inference Tree – Example of the 2/3 Majority Voter

After having given a generic description of the IT, we illustrate the development of the inference tree through an example of a majority voter. Consider a triple modular redundant (TMR) system, where three process replicas produce results for a voter to generate a majority response. Request ordering is a critical issue, that is, we want all replicas to process the same sequence of requests. One way to handle this is to allow each client to attach a timestamp to each request. Another key issue in the voter is that of vote synchronization, i.e., ensuring that the tabulated result is based on a set of votes that are all responses to the same request. Communication delays or other problems may prevent some votes for a particular request from reaching the voter in a timely manner. As we do not impose any constraints on the voter itself, the voter must rely on other information for synchronization. Moreover, a voting session takes place whenever there are sufficient number of votes for a given failure class (e.g., fail-stop) for a particular request. If a replica's vote misses its intended voting round, the vote is treated as an obsolete vote.

Fig. 2 represents the generation of an IT for a majority (2/3) voter. Each node of the tree represents a primitive FUNCTION (or functional block of the protocol) at a given level of abstraction. Here, FUNCTION is the 2/3 voter, i.e., 2 out of 3 nodes need to agree for a result.

Further, in Fig. 2, a set of CONDITIONALS $C[\cdots]$ describes the various conditions (actual or speculative) imposed on the voter. As examples, $C[Time\_Window]$ indicates a condition that a message will be processed by the voter only if it arrives in a specified time window, say $[t-\Delta, t+\Delta]$, $C[Sequence]$ indicates the condition on the sequence of message arrival, $C[Count]$ denotes the number of votes received for a particular round, and $C[Round]$ imposes constraints that all the messages are

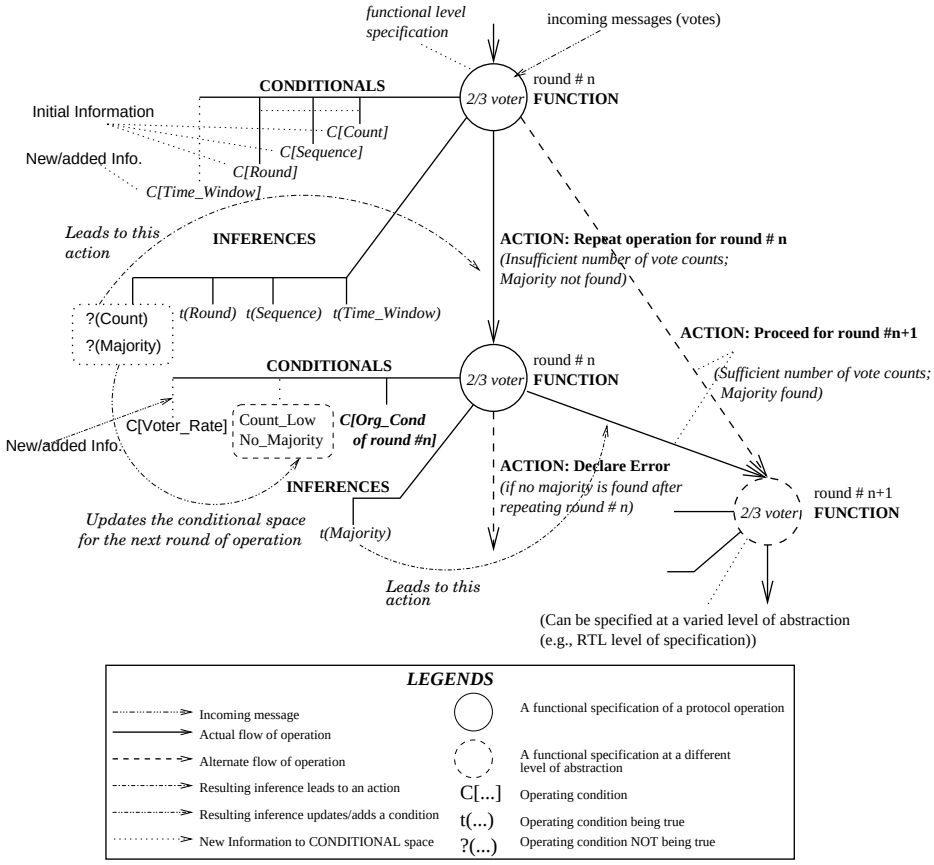

Fig. 2. The Inference Tree for a 2/3 Voter Protocol

from the same round n. Based on the inputs to the voter and the governing conditions mentioned above, specific ACTIONS such as the voter outputs a result (and proceeds to the next round) or a repeat of the voting process, and corresponding operational INFERENCES are generated. In the INFERENCE space, $t(Round)$ denotes that the condition $C[Round]$ is true whereas $?(Count)$ reflects the fact that the condition $C[Count]$ is not satisfied.

Based on inferences, a specific action is taken. These resulting inferences in turn get reflected in the CONDITIONAL space of the IT depicting information for the next round of operation, to govern the subsequent rounds of protocol operation. In Fig. 2, we also highlight which inference(s) leads to which action(s) (depicted with arrows labeled "leads to this action") as well as which resulting inference(s) causes updating of the conditionals for the subsequent round of protocol execution (depicted by arrows labeled "updates ... operation"). Note that based on the prior inference (first instance of round # n) of $C[Count]$ not being satisfied, during the second instance (repeat) of round # n if sufficient number of votes are not received, then an action such as "Declare Error" could be taken. $C[Org\_Cond\ of\ round\ \#\ n]$ captures all the conditions that were imposed during the first instance of round # n.

ACTIONS are protocol-related. For example, for a 2/3 voter as depicted in Fig. 2, we outline two potential ACTIONS that could be taken after round # n. If a sufficient number of votes and all other related conditions were satisfied, the voter proceeds with the next round of voting process, otherwise, the voter may repeat the operation for round # n. These can be considered as branching points where the protocol takes a decision based on information it has gathered.

A novel property of the IT structure is that it allows for refinements in specification. Initially, the IT representation is at the protocol level. Over subsequent iterations, parametric/implementation information is added. For example, in Fig. 2, in the CONDITIONAL space of the IT depicting the second instance of round # n activities, a condition $C[Voter\_Rate]$, indicating TMR voting rate to be greater than or equal to the message input rate, can be added as an implementation detail (beyond the traditional descriptions of TMR) to the specification. As new conditional or parametric information is incorporated, a complete verification (and inference) cycle is performed to highlight any inconsistency the new parameters might

generate. It is of interest to note that the conditional and inference space is dynamically re-generated over each round of verification. Moreover, as we only functionally enumerate the operations of a protocol, the size of the IT is bounded by the inference space and actions. Thus, each stage of IT refinement only linearly adds more parameters in the CONDITIONAL or INFERENCE space. For example, adding a conditional of "timing" to the 2/3 voter results in a consequent inference list that enumerate the list of operations on/from which "timing" could have a potential effect on the IT.

Although, the IT visually outlines the protocol operations, it does not (in itself) provide any FI related information. However, the deductive capabilities of formal methods permit us to pose queries and identify the dependencies based on the verification information acquired within the IT structure. The DT structure, described next, utilizes the IT generated inferences to facilitate query mechanisms to identify FI test cases.

Dependency Tree (DT): Query Engine

Deductive logic used by the verifier is applied to determine the actual dependency of the function on each individual variable, thus determining the actual subset of variables that influence the protocol operation. The DT is generated by identifying all functional blocks of a protocol, and ascertaining the set of variables (also function variables) that directly or indirectly influence the protocol operation. The set of conditions in the IT (appearing the CONDITIONAL space) forms the initial set of variables in the DT. This initial set of conditionals serve as an actual (or speculative) list of variables for the DT. If the verification process at a particular level of abstraction completes successfully, as per our intended objectives, we make use of the DT to identify the list of assumptions, variables and functions on which the overall protocol operation or a specific aspect of the protocol operation depends on. Pertinent information for these dependencies are essentially captured in our IT structure. This dependency list along with constraints (conditionals) is then passed on to the test cases generation tool to construct specific tests for FI experiments. On the other hand, if a conflicting condition is flagged and gets reflected in the IT INFERENCE space, we initiate deductive reasoning through the DT. The DT allows queries[2] about the protocol behavior to be posed following the inconsistency to determine the dependency over certain variables i.e., we try to uncover the reason(s) that causes the inconsistency. If the "inconsistency" is dependent on a given set of variables, then we can inject faults into these variables to observe the behavior of the protocol in such faulty cases.

In case a protocol involves operations over multiple rounds, the corresponding DT also is iteratively generated over rounds. At each iteration, the dependency list is pruned as one progresses along a reachability path. In the absence of any new conditionals being added, the dependency list of the DT is monotonically decreasing. In case new conditionals are specified, variables which were pruned earlier from the dependency list may reappear in the next DT iteration. The leaves of the tree represent the minimal set of variables that are associated, or provide influence[3] on the operation of each primitive function of the protocol.

Fig. 3 depicts a general working of the DT for a round-based protocol and highlights key processes involved. The actual dependency of the function $P(n)$ on individual variables, assumptions, etc. as determined by the verifier is stored in some form of a database. The actual or speculative list of variables or conditionals as specified and captured in the IT (CONDITIONAL space) forms the input for querying the dependency of the function on them. The output of a query provides the dependency of the protocol on either variables or conditionals. Inferences and associated actions taken at a round link the DT process at the next round of protocol operation. In case new information has been introduced, query output would produce a refined list indicating dependency on newly added variables/conditionals. At the terminal round, the DT process provides a complete dependency list of variables/conditionals required for ascertaining correctness of a specific property of the protocol. Different pairing/combinations and orderings of variables appearing in this identified list constitute distinct fault-injection experiments. Note that axioms and theorems required for establishing the correctness of a specific property of the protocol are important inputs for formulating FI experiments, as these sets of statements provide insights to basic conditions which need to be validated in an implementation also.

Next, we illustrate how the DT for a 2/3 voter can be processed (Refer to Fig. 4). Based on the information captured in the IT (See Fig. 2), in order to identify key variables and parameters, we initiate the query processing mechanism in the DT. For round # n activities, we evaluate the dependency of different assumptions and

[2]It is to note that queries in the DT's can be formulated as (a) conjectures and posed to the theorem prover of the underlying formal engine to ascertain dependencies of the protocol operation on certain variables or (b) simple database operations to retrieve list of variables from the tables storing verification information.

[3]In case dependencies in the protocol arise due to subtle lower level details which have not been specified, then naturally these dependencies will not be uncovered. It is important to consider that the "completeness" of the variables set is complete only to the "level of specification" actually specified.

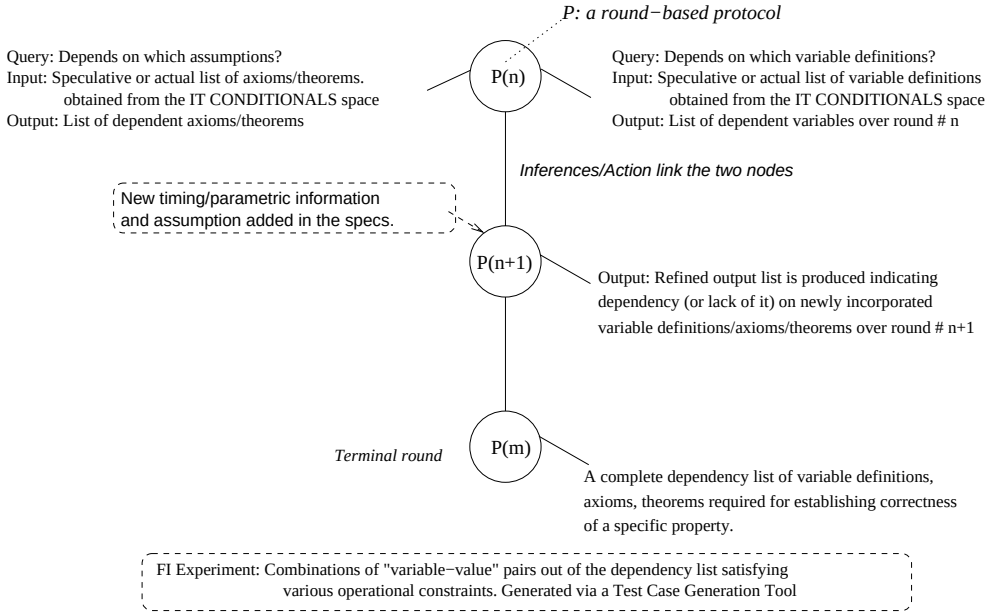

Fig. 3. The Dependency Tree : Highlights of Key Processes Involved in a Round-based Protocol

variable definitions by parsing the information generated over the verification process. In Fig. 4, predicate $voted?$ returns true if the given replica voted, $vote\_ok?$ returns true if the vote is not obsolete, and *fail-stop_maj_ok?* returns true if sufficient non-obsolete votes are there for finding majority. Note that the DT points out that the chosen implementation of the 2/3 majority voter does not depend on $C[Sequence]$. For other fault-tolerant majority voting schemes such as a function which discards top k and bottom k values and then takes the median of the remaining values, the correctness of such a voter depends on the sequencing of the requests as governed by $C[Sequence]$.

We emphasize that the DT may not fully represent all possible variable dependencies as it will always be limited to the amount of operational information actually modeled into the formal specification. At any desired level, the elements of the current dependency list provides us with a (possibly) minimal set of parameters which *should* help formulate the FI experiments via all permutations and combinations, and *ideally* should generate specific (or a family of) test cases. We repeat that our intent is pre-injection analysis in identifying specific test cases. The actual FI experiments are implemented from these test cases based on the chosen FI tool-set(s).

C. Overall Process of Identifying the Influential Set of Protocol Variables/Conditions

In order to realize our proposed methodology for pre-injection analysis, we have used PVS specification language to specify the protocol operation and its theorem prover to establish the correctness of various properties of interests. The construction of IT/DT and subsequent analysis in the DT as discussed earlier is essentially carried out by exploiting the information that gets generated as part of verification process. The derived dependency-list gets stored in the DT and subsequently used to perform certain queries for pre-injection analysis.

In order to prune the list of variables (and in turn state-space associated with them), we compare the list provided by the DT process with the actual or speculative list of variables/conditional specified in the IT. Utilizing the DT information and comparison results, we identify the redundant variables and/or conditionals specified/used in the initial specification of the protocol. These redundant variables (those variables that are specified but are not influencing in anyway the protocol operation) are then eliminated from the IT CONDITIONAL space and the verification process is repeated again to ensure that the specification and the corresponding verification are consistent and up-to-date. Next, test cases for an FI experiment for a chosen tool-set can be constructed using the identified minimal set of variables.

D. Generation of Test Suites for Fault-Injection Experiments

In order to support the test generation aspect of our proposed methodology, we have developed a tool called Sampurna [56] which generates a comprehensive set of test suites by eliminating the variable-value pairs that are not attainable/possible with respect to the protocol specification by using *a priori* knowledge of the system. The concept of cross product is introduced to capture all the possible combination of variables so as to generate

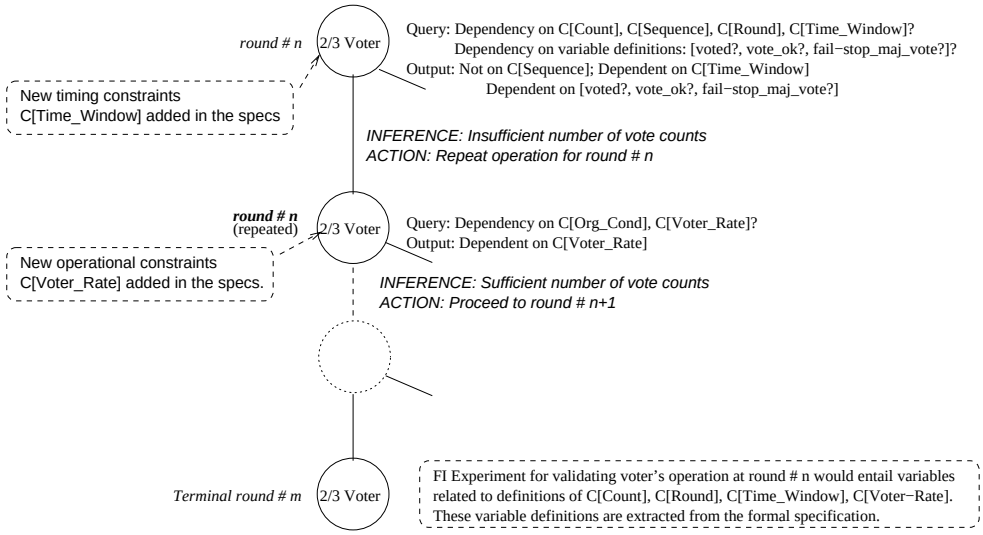

Fig. 4. The Dependency Tree : 2/3 Majority Voter

set of test case scenarios. The constraints are applied over this cross product to restrict the irrelevant test cases thus achieving comprehensiveness and still satisfying test coverage. After obtaining the final constraint-cross product, based on *a priori* knowledge of the working principle of the protocol, the redundant and irrelevant test cases are being removed. The expected output of the tool is test cases containing variables and their associated values that would steer the system through different states so as to detect any discrepancies with respect to the expected correct behavior of the protocol.

The Sampurna tool utilizes the dependency list obtained in the DT to generate test cases for guiding the FI-based validation. The steps of the test cases generation procedure are as follows:

Step I: Assimilate the complete (or a part of; based on user's intuition) set of variables and their associated values/ranges. These variables are part of a minimal set of variables on which a particular stage of the protocol operation depends on.

Step II: Eliminate redundant and unattainable test cases using the information captured in the IT conditional space and/or *a priori* knowledge of the protocol operational behavior.

Step III: Reduce further the number of the resulting test cases by applying additional constraints, if there are any, that a user may impose on the system.

In Sampurna, variables identified by the DT are stored in different tables depending upon their functionalities and queries are formulated considering the tables as its input and using logical relations among variables. Multiple queries could possibly be formulated to generate the desired set of test cases. The final output of these queries are stored in the table and reports can be generated to be used by a tester or user of the system.

After having described the overall IT/DT based approach for generating FI experiments, we now present a case study of a basic online diagnosis protocol (hereafter referred as the WLS Algorithm) introduced in [58], where we highlight the construction of IT and DT structures for the same, and discuss how relevant test cases were generated to validate an implementation of this diagnosis algorithm against these specific tests.

IV. Pre-Injection Analysis for FI-Based Validation of the Online Diagnosis Protocol

A. An Overview of the WLS Algorithm and its Formal Specification and Verification

In [58], authors have presented comprehensive online diagnosis algorithms capable of handling a continuum of faults of varying severity at the node and link level. The WLS algorithm which deals with node (benign) faults utilizes a two-phase diagnostic approach: **phase 1:** local syndrome formulation based on a node's local perception of other nodes; this is based on that node's analysis of incoming message traffic from other nodes, and **phase 2:** global syndrome formulation through exchange of local syndrome information to all other nodes. In subsequent discussions, terminologies and algorithm description are taken directly from [58].

Terminology

Let N be the number of processors in the system and $mess_j$ represent a message sent by processor j. As the communication model is frame based with messages sent/received by nodes at the frame boundaries, the frame number is also a useful component in identifying a message. Let $\mathscr{M}_i^n(j)$ define the set of all $mess_j$ received by processor i as composed/sent by j during frame

n. Fault categories for the messages are based on the receiver's observations on these messages. Two such fault categories are: (a) The set of *missing messages*, $MM_i^n(j)$, are those messages which i believes j failed to issue during frame n, and (b) The set of *improper logical messages*, $ILM_i^n(j)$, are those messages which are correctly delivered but disagree with V_i, the result of i's own voting process on inputs received. The *syndrome* $\mathscr{S}_i^n(j), \forall i, j$ represents the union of $ILM_i^n(j)$ and $MM_i^n(j)$. $\mathscr{S}_i^n(j)$ is represented in vector form for each value of i, with vector entries corresponding to all j values from which i receives messages. The vector entry corresponding to any node j is a binary input: 0 corresponding to a fault-free input received from j as perceived by i, and 1 for a fault being perceived by i.

Each node maintains its perception of the system state using a system level error report, $F_i^n(j)$, consisting of an ordered quadruple $\langle i, j, n, \mathscr{S}_i^n(j)\rangle$. The function $F_{tot}^n(j) = |\bigcup_{i\in N, i\neq j} F_i^n(j)|$ is used to count the number of accusations on processor j by all other monitoring processors during frame n. Thus, $F_{tot}^n(j)$ is an integer where $0 \leq F_{tot}^n(j) \leq (N-1)$.

Diagnosing Benign Faults

The *processor–processor (PP)* model assumes that all the communication links are non-faulty and that processors are the only potentially faulty units.

Algorithm PP (WLS)

D1.0 For all $i, j \in N$, each processor i monitors each $mess_j \in \mathscr{M}_i^n(j)$.
- D1.1 If the value v_j contained in $mess_j$ does not agree with V_i, then $mess_j \in ILM_i^n(j)$,
- D1.2 If $mess_j$ is missing, then $mess_j \in MM_i^n(j)$,
- D1.3 Update the syndrome information: $\mathscr{S}_i^n(j) = ILM_i^n(j) \cup MM_i^n(j)$.

D2.0 At the completion of frame n, for every j, each i will determine if an error report should be issued: if $\mathscr{S}_i^n(j) \neq \emptyset$ then send report $F_i^n(j)$ (as composed/sent by i) to other processors, else do not send $F_i^n(j)$.

D3.0 For each j, as frame $n+1$ completes, compute $F_{tot}^n(j)$.
- D3.1 If $F_{tot}^n(j) \geq \lceil N/2 \rceil$ then declare j as faulty.
  - D3.1.1 If processor k failed to report $F_k^n(j) = \emptyset$ then $mess_k \in MM_i^{n+1}(k)$
- D3.2 If $F_{tot}^n(j) < \lceil N/2 \rceil$ then
  - D3.2.1 If k reported $F_k^n(j) \neq \emptyset$ then $mess_k \in ILM_i^{n+1}(k)$

D4.0 Increment frame counter n and proceed to step D1.

The error detection process is summarized by step D1.0. During frame n, each processor monitors the messages received and performs error checking. The logical content errors identified in step D1.1 are detected by voting on the inputs and then checking the inputs against the voted value (i.e. deviance checking). Omissions of expected messages are also detected and recorded in D1.2. In step D1.3, these errors are written into a local error log to be processed at the completion of frame n. In step D2.0, if any errors have been logged, a system level report is issued accusing the suspected processor. These reports are counted in step D3.0 and the accused processor is declared faulty provided at least half of the system agrees on the accusation. The diagnostic processors are also checked as part of the algorithm. In D3.1.1, if j is determined to be faulty but a monitoring processor k failed to report an error on j, processor k will be accused as faulty in the succeeding round of diagnosis. In D3.2.1, if only a minority of processors accused j, they will be accused as faulty in the next round.

Formal Treatment of Algorithm PP (WLS)

In order to facilitate formal analysis, in [58] the authors have simplified the algorithm emphasizing the operations being performed and the properties that are needed to be formally specified and verified. The simplified form is as follows:

PP(0)

1) All accusations of faults are cleared.

PP(n), n $>$ 0

1) Each processor i executes one frame of the workload, arriving at some value $Val^n(i)$.
2) Each processor sends $Val^n(i)$ to all other processors.
3) Each processor i compares incoming messages to its own value:
   a) If the value from j does not match, is missing, or is otherwise detectably benign, or there is an accusation from the last frame of i against j, i records that j is BAD.
   b) Otherwise, i records that j is GOOD.
4) Each processor sends its report on each other processor to all processors.
5) Each processor i collects all votes regarding each other processor j:
   a) If the majority of votes are BAD, then processor i declares j faulty. Furthermore, i records an accusation against any processor k that voted j GOOD.
   b) If the majority of votes are GOOD, then i records an accusation against any processor k that voted j BAD.

In this rewriting of the algorithm, the initial frame, referred to as PP(0), simply initializes the data structures appropriately. Next, a workload frame is executed (Step 1), arriving at some value, Val. Processors then exchange values (Step 2). All good processors should then have exchanged identical values. Faulty processors may have exchanged corrupted values that are locally detectable; the possibility of faulty processors delivering different values to different receivers is not considered. All processors then compare the exchanged values with their own. Any discrepancy is recorded as an accusation against the sending processor.

Developing the Formal Specification of PP

The formal specification of PP is specified in a single PVS theory called *pp*. In the theory *pp*, some other predefined theories are explicitly imported[4]. This theory takes several parameters which include m, the maximum number of periods, n, the number of processors, and T, the type values that are passed between processors. The term *error* represents values that are benign upon local receipt, such as missing values, values failing parity check, values failing digital signature checks, and so on. *BAD* and *GOOD* are the values of accusations sent by processors over the network. Finally, the function *Val* is assumed to return the correct value for each frame of computation, and that the correct value is never any of the special values *error*, *BAD*, or *GOOD*.

The type *statuses* is defined to be an enumeration of three constants, corresponding to three of the categories of behavior: *symmetric-value faulty*, *benign*, and *good*. The function *status* returns the status of a given processor (or fault containment unit *fcu*).

Some notations are used for describing statuses: *s*, *c*, and *g* are predicates recognizing the symmetric-value faulty, benign, and good processors, respectively. Similarly, given a set *caucus*, *as*(*caucus*) is the set of arbitrary-faulty processors in *caucus*. The functions *ss*, *cs* and *gs* similarly select the symmetric-value faulty, benign, and good processors, respectively.

The function *send* captures the properties of sending values from one processor to another. This function takes a value to be sent, a sender, and a receiver as arguments; it returns the value that *would be* received if the receiver were a good processor. The behavior of *send* is axiomatized according to the status of the sender. The first axiom simply says that a good processor sends correct values to all (good) receivers: $g(p) \supset$ *send*(t,p,q)$= t$. The second axiom says that a benign faulty processor always delivers values that are recognized as erroneous by good receivers: $c(p) \supset$ *send*(t,p,q) = *error*. The third axiom says that a symmetric-value faulty processor sends the same value to all good receivers, although that value is otherwise unconstrained (i.e., it may be any possible value, including those that are recognized as erroneous) $s(p) \supset$*send*(t,p,q) = *send*(t,p,z). Nothing is specified for the behavior of asymmetric-value faulty senders. A lemma (called *send5*) is stated and proved that all receivers obtain the same value no matter what the status of the sender (here, the possibility of link and arbitrary faults is discounted) *send*(t,p,q) = *send*(t,p,z).

The function *HybridMajority* is intended to be similar to the standard *Majority* function, except that all *error* values are excluded. The function *HybridMajority* takes two arguments, a set of processors (i.e., an *fcuset*), which we call the *caucus*, and a vector mapping processors to values (i.e., an *fcuvector*). Several properties related to *HybridMajority* that are of particular interests are described below:

The first property states that if the vector records the same non-error value for all good processors in the caucus, and the vector records an error value for all benign-faulty (benign) processors in the caucus, and there are more good processors than symmetric-value faulty processors in the caucus, then *HybridMajority* returns the same value as that recorded in the vector for the good processors.

The second property states that the value returned depends only on the values recorded in the vector for the processors in the caucus.

The final property deals with the fact that if there are more good than symmetric-faulty processors and all good processors agree on some non-error value, and the *HybridMajority* function returns a value, then that value is the value of each good processor.

Next, the definition of some of the key functions of the actual algorithm is discussed.

```
Syndrome(R,j,i,OldAccuse):T =
 IF OldAccuse(i,j) OR (NOT Val(R)=
   send(Val(R),j,i)) THEN BAD
 ELSE GOOD
 ENDIF
```

The *Syndrome* function above is meant to capture the property that in period R, i believes j is faulty. The parameter *OldAccuse* essentially records old accusations from earlier periods. The only other reason to accuse a processor of faulty behavior is if that processor sent some value that does not correspond to the correct value. The next function `KDeclareJ` (i.e., k declares j faulty) is built using the *Syndrome* function. The definition is:

```
KDeclareJ(pset,R,OldAccuse,j,k):bool=
 HybridMajority(pset, LAMBDA i:
  send(Syndrome(R,j,i,OldAccuse),i,k))=BAD
```

This predicate is meant to capture the idea that processor k will gather all accusations against some processor j, and then take the *HybridMajority* of that set. If most processors accuse j, then this predicate is true, i.e., k declares j faulty. The main function, for "processor-processor model" based diagnostic algorithm, *PP* is specified below:

```
PP(pset,R,OldAccuse)(i,j):RECURSIVE bool =
IF R=0 THEN FALSE
 ELSE KDeclareJ(pset,R,OldAccuse,j,i) OR
  PP(pset, R-1, (lambda i2,k: OldAccuse(i2,k) OR
   EXISTS j2:(KDeclareJ(pset,R,OldAccuse,j2,i2)/=
    (send(Syndrome(R,j2,k,OldAccuse),k,i2)=BAD))))(i,j)
ENDIF MEASURE (LAMBDA pset, R, OldAccuse : R)
```

The intended meaning of this formal description is that after R periods, starting with *OldAccuse* accusations,

[4]The complete theory specification adapted from [58] is presented in the Appendix.

processor i believes that processor j is faulty. The function *PP* is defined as a recursive function. If the number of periods *R* is zero, then i will not accuse j. If *KDeclareJ(pset,R,OldAccuse,j,i),* that is, if after gathering votes for period R, a (hybrid) majority of other processors send i an accusation of j, then i believes j is faulty. Otherwise, *PP* is called recursively, using one less period. The recursive call also updates *OldAccuse* to include the case that some processor misdiagnosed some other processor. That is, an accusation is added to the local *OldAccuse* for the next period if the voted diagnosis *KDeclareJ(pset,R,OldAccuse,j2,i2)* of some processor $j2$ does not agree with the individual accusation sent from k to $i2$.

The two properties dealing with *soundness* and *completeness* are formally specified and verified using PVS in [58]. We have added (and at places modified) a few specifications as needed. The first requirement that of *Soundness* states if the algorithm `PP` declares a processor to be faulty, then it is indeed faulty. The key property addressed here is that all good processors accuse only faulty processors of being faulty. Essentially, we want to prove that if i is good, and after R periods of PP, i accuses j, then either j is benign or symmetric-value faulty. The second property, *Completeness*, states that if a processor is faulty, then `PP` will determine this.

B. Visualization: IT/DT for the WLS Fault-Diagnosis Algorithm

The formal verification of the two properties stated above is based on the prove-by-induction on the number of rounds. The PVS tool allows the user to conduct partial proofs under different assumptions and special cases of interests.

The objective of the formal verification and representation of verification information in the IT structure is to guide the selection of appropriate queries to be posed in the DT. It is important to note that the selection and formal representation of queries to be posed is still an interactive process. This is typical for any theorem proving (proof theoretic) environment where the user's knowledge of the specified protocol activities guides the process of query formulation. Note that for both IT and DT, we describe them in simple English as depicting the information in the formal syntax of PVS would not be appropriate for general readers.

Development of the IT Structure

In Fig. 5, we depict the operational flow of the PP (WLS) algorithm for a particular node for three rounds of activities starting with round # n. The initial set of conditionals on which the protocol operation begins with is listed below.

- $g(p) \rightarrow send(t,p,q) = t$
- $c(p) \rightarrow send(t,p,q) = error$
- $s(p) \rightarrow send(t,p,q) = send(t,p,z)$
- $send(t,p,q) = send(t,p,z)$
- $\forall p : g(p) \wedge p \in caucus \rightarrow v(p) = t \wedge t \neq error \quad [\mathscr{A}]$
- $\forall p : c(p) \wedge p \in caucus \rightarrow v(p) = error \quad [\mathscr{B}]$
- $\|caucus\| = \|cs(caucus)\| + \|ss(caucus)\| + \|gs(caucus)\|$
- $\|gs(caucus)\| > \|ss(caucus)\| \wedge \mathscr{A} \wedge \mathscr{B} \rightarrow$ *HybridMajority*$(caucus, v) = t$
- $N \geq 3$ and $E < \lceil N/2 \rceil$ where N and E are the total number of nodes and the number of faulty nodes, respectively.
- $Syndrome_i^n(j) = BAD \rightarrow \neg(Val^n(j) = send(Val^n(j), j, i)) \vee OldAccuse(i,j)$

As a general rule, to guide the proof process to proceed in a desired way, we add conditions as the proof-steps are taken. For processor i to judge processor j in round # n, it looks at either the value sent by processor j (i.e., *send*$(Val^n(j), j, i)$) or an old accusation about processor j (i.e., *OldAccuse*). By setting the predicate *OldAccuse(i,j)* to be true, we let the function *PP* to return true by setting the predicate *KDeclareJ* to true over the round # $n+1$. *KDeclareJ* being true indicates that after n rounds, starting with *OldAccuse* accusations, processor i believes that processor j is faulty.

Similarly, for a processor k to be declared faulty by processor i over round # $n+2$ as it could not diagnose processor j to be faulty as majority of processors did declare j to be a faulty processor, in the recursive part of *PP* with one less round (i.e., for round # R-1), the second clause (that is, `EXISTS j2...` appearing in a snippet of formal specification of PP) need to be set true in order to update *OldAccuse* to reflect that processor k misdiagnosed processor j.

We now describe the IT for the WLS algorithm depicting the operational flow for a node 'i' in the system (See Fig. 5). The ways of triggering or setting various conditions to steer the flow of protocol operation have been discussed in the preceding paragraphs. During the execution over round n, node i receives a message from node j and also a syndrome of j from node x as prepared by it after round # $n-1$. $C[Set]$ in the *CONDITIONALS* space reflects the initial set of conditions. Over round n, based on the value received from node j and a syndrome from node x reflecting that it suspects j to be faulty, node i suspects j to be faulty, informs other nodes about its assessment and then proceeds to the next round. These inferences have been captured in the *INFERENCES* space. They in turn update the *CONDITIONALS* space for the next round ($n+1$) and also lead to the specific action of recording j 'BAD' and sending a report. Based on the notations introduced in Fig. 2, we have highlighted these in Fig. 5 with arrows labeled "Updates..." and "Leads to...", respectively. Over round # $n+1$, based on the reports from other nodes about node j after round n, node i collates this information and performs the majority voting. If the majority of nodes

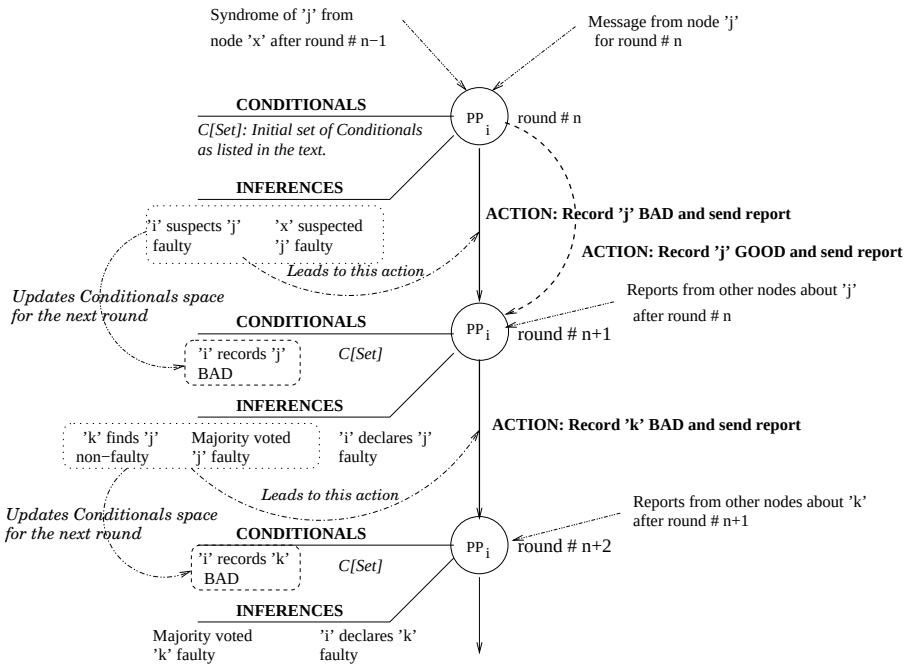

Fig. 5. The IT for the WLS Online Diagnosis Algorithm – Operational Flow Illustrated for Node 'i'

voted node j to be faulty, then node i also declares node j to be faulty. If a node k fails to find j faulty, then node i prepares a syndrome for node k and sends that to other nodes. Over round # $n+2$, based on the reports from other nodes about node k after round $n+1$, node i collates this information and performs the majority voting. If majority of nodes found node k to be faulty, then node i also declares node k to be faulty. In the event, if one of the conditions were not satisfied, alternate actions could have been taken as marked in Fig. 5.

Development of the DT Structure

In Fig. 6, we illustrate how the DT of the WLS algorithm can be processed. Based on the information captured in the IT (Fig. 5), in order to identify key variables and conditionals, we initiate the query processing in the DT. For round n activities, we determine the actual and lack of dependency on the conditionals/variables as listed in the CONDITIONAL space of the IT. At each iteration, the dependency list is pruned as one progresses over multiple rounds of protocol execution. Moreover, in case new conditionals are specified, variables which were pruned earlier from the dependency list may re-appear in the next iteration. As illustrated in Fig. 6, round n of the protocol operation does not depend on assumption *HybridMajority*, however, upon adding *timeout* as a new condition for the subsequent rounds of operation, the assumption *HybridMajority* re-appears in the dependency list for rounds # $n+1$ and $n+2$. Below we highlight the complete list of dependencies for the *completeness* property *PP* (i.e., if a processor is faulty, then the PP will determine this) to hold.

> **Dependency List:** *s, g, c, gs, cc, ss, send1, send2, PP, Empty, HybridMajority, KDeclareJ, Syndrome, OldAccuse*

FI experiments for validating the PP (WLS) algorithm at rounds $n, n+1$ and $n+2$ would entail variables related to the definition of the terms listed above. We provide further details on this aspect in Section IV-C where we discuss validation of a Java implementation of the WLS algorithm.

C. Validation of a Java Implementation of the WLS Algorithm

We have implemented the online diagnosis algorithm (PP) in Java. A requirement was that a user can verify if a processor was declared as being faulty by monitoring the outputs on the command line. Instead of a processor receiving values for a workload and computing majority to obtain a value, we decided that each node would send only one value per frame. This helps in determining when to end a frame, as we would wait for a defined time period in order to receive messages from all the other processors in the network. Also, as per the original description of the algorithm if there were no errors then there would be no error reports sent. However, there is no specification on how long a processor should wait for an error report. So, we included a timeout so that error reports received before the timeout would be processed

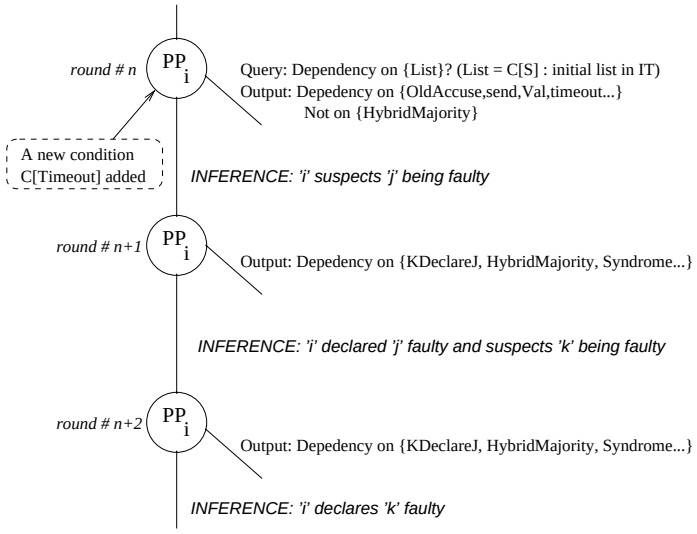

Fig. 6. The DT for WLS Online Diagnosis Algorithm – Process Illustrated for Node 'i'

and error reports received afterward would be considered as being lost message and gets discarded.

As the protocol operation essentially depends on assumptions relating to *sendOldAccuse*, *Syndrome*, *caucus* and *HybridMajority*, the key test cases for three specific rounds of operations are generated using the Sampurna tool. The description of fault-injection scenarios to be executed is given below:

- For round # n
  - Corrupt the variable containing *Val*.
  - Delay the message containing *Val* to get it recorded as a *missing* message.
  - Corrupt the variable containing *Syndrome*
  - Corrupt the variable containing *OldAccuse*
  - Delay the message containing error report to force it to get accused in the subsequent round.
- For round # $n+1$
  - Corrupt the variable containing $F^n_{tot}(j)$.
  - Even if $F^n_{tot}(j) \geq \lceil N/2 \rceil$, corrupt the variable containing processor k's *Syndrome* generated with respect to j; that is change *OldAccuse* for the next round.
  - Even if $F^n_{tot}(j) \leq \lceil N/2 \rceil$, corrupt the variable containing processor k's *Syndrome* generated with respect to j; that is change *OldAccuse* for the next round.
  - Increase the number of faulty processor such that the condition $\|gs(caucus)\| > \|ss(caucus)\|$ no longer holds.
- For round # $n+2$
  - Corrupt the variable containing $F^{n+1}_{tot}(j)$.

The parameters *Val*, *Syndrome*^{n}, *Syndrome*$^{n+1}$, *OldAccuse*, and *HybridMajority* take Boolean values. $F^n_{tot}(j)$ and $F^{n+1}_{tot}(j)$ can have the value either more or less than the majority value. Three combinations relating variables *Val*, *OldAccuse* and *Syndrome* are not attainable, e.g., a combination such as *Val* being *false*, *OldAccuse* being either *true* or *false* and *Syndrome* being *false* is not valid. Subsequently, we have a set of 21 FI scenarios. Further, two delay operations and a case causing the number of *good* processors to be less than that of *faulty* ones result in a total of 24 tests for fault injection experiments.

Our Java implementation of the WLS (PP) algorithm was subjected to a total of 24 test cases, and we were able to identify 3 software design faults that were causing the program to not get executed as per the specified requirements. We describe these findings below:

- One of the design fault had to do with the omission of 'timeout' notion in our initial specification that was causing the program to wait for an arbitrarily long time for either the message or an error report to arrive. This case was simulated by inserting a perturbed_delay function that permits us to selectively delay, or fail to delay, at the point where it is inserted.
- Related to the previous finding, via the perturbed_delay function, we also discovered a synchronization problem which was caused by having the processor do the sending, receiving, and processing of messages thereby leading to concurrency issues in the case when the processor got interrupted while processing a message. Later on we rectified this problem by using a message container for the messages to which both the processor and threads responsible for receiving and delivering the message to the processor would read and write.

- Another interesting case that revealed the deficiency in our implementation was that we had missed out the checking of the processor's health to determine whether it is a healthy or faulty processor. At times, in our system the number of faulty processors would be more than the good or healthy processors and still we would perform majority and take the votes of faulty processors to get a majority vote. This sometimes led to declaration of a healthy processor as a faulty in a caucus of three processors. From our viewpoint, this case would not have been identified via classical testing techniques. This became a trivial test to conduct as the dependency list included terms gs and ss, and the condition $\|gs(caucus)\| > \|ss(caucus)\|$ was also captured in the IT conditional space.

Discussions

Formal methods require the right mix of effort, expertise, and knowledge. In this particular case study, we leveraged from the work already done as part of the development of online diagnosis algorithms [58]. However, in our other case studies [51], [49], substantial efforts (about 1 man-year) were put in developing the formal specification and subsequently verifying desired properties of the protocols used therein. It is important to emphasize that nowadays protocols developed for highly dependable systems typically go through formal verification, and it would be ideal to exploit (and reuse in a meaningful way) the information generated over the verification process to guide the validation of an implementation of that protocol.

V. Comparative View with Related Work

The classical use of formal methods has been for the verification of protocols, and specifically, on finding design stage flaws in the protocols. In [6], the focus of the work is on the verification of fault-tolerance properties using model-based formalisms, specifically an executable specification has been developed to establish the tolerated behavior of the spacecraft computers in presence of faults. In literature, a variety of approaches have developed excellent concepts in linking formal approaches to testing (See [2], [4], [7], [8], [12], [16], [23], [28], [29], [35], [39], [52], [55], [60] among others). While there has been a lot of work on specification-based testing and test case generation [15], [22], not much work has focused on bridging the gap between theorem proving and testing. In [13], authors have presented the HOL-TestGen system that generates unit tests from Isabelle specifications. In [9], a tool that uses HOL specifications for testing protocols has been discussed. In [41], the author has presented strategies to create random test cases directly from PVS [40] specifications.

In particular, to the best of our knowledge, the key distinction of our approach from others is that we make prominent use of proof-theoretic-based reasoning, and link/analyze the inferences generated over the verification process to determine key assumptions and the set of (implementation) parameters to derive scenarios to drive FI experiments. Though our approach is proof-theoretic, we could potentially utilize (and interface) model theoretic approaches as well. In [45], the authors have developed a unified framework to provide support for both proof-theoretic as well as model-theoretic approaches. As mentioned in Section III-B, our approach allows for mixed level of abstraction. For example, at the circuit level abstract, a function can be modeled in say RTL-level specification. Such a low-level abstraction of the program is useful to reason about hardware errors. The formal model can then be rigorously analyzed under error conditions against the above specifications using techniques such as model checking and theorem proving.

Existing efforts [1], [5], [10], [16], [18], [19], [21], [53], [54] have explored deterministic approaches for test case identification for validation. The work reported in [16], [19], [21] have exploited some typical properties of fault tolerant protocols (e.g., decision stages, chain of furcated fault-handling actions, etc.) for modeling complex distributed protocols. Our proposed representation schemes share properties with other state-transition representations like assertion trees [5], [54] or Petri nets [20]. We point out that [20] uses a formal specification of the protocol and it is processed using some heuristic to identify influence parameters in an automated manner. In particular, reachability analysis is performed to identify fault cases and their corresponding activation paths. In order to reduce the size of the reachability graph certain restrictions on the protocol behavior is assumed. This scheme works well for bounded systems, however, for protocols dealing with real-time and non-deterministic attributes, this approach is limited.

Similar to our proposed approach, the work reported in [33] presents a symbolic approach for injecting faults (those relating to HW errors only) into programs written in Java and considers the effect of bit-flips in program variables. However, HW errors which can alter the control flow of the program have not been considered by the technique. In [44], the authors have presented a program level framework that allows specification of arbitrary detectors and their verification against transient HW errors using symbolic execution and model checking. In a recent paper [26], the authors have proposed a framework for generating test vectors from specifications written in the Prototype Verification System (PVS) [40]. The methodology uses a translator to produce a Java prototype from a PVS specification. Symbolic (Java)

PathFinder [43] is then employed to generate a collection of test cases. The combination of these two existing tools enables this process by automating much of the task.

VI. Conclusions, Summarizing Past Work and Future Directions

The conventional FI approaches are facing growing limitations in handling the large state space involved in the operations of dependable distributed and real-time protocols. We have shown the efficacy of formal techniques as an supplement for FI-based validation of dependable distributed protocols that have a formal specification and whose models have been validated. In this paper, we have applied our approach to an online diagnosis algorithm and illustrated the effectiveness of the proposed pre-injection analysis in identifying relevant though critical test cases against which an implementation of the diagnosis protocol must be validated.

In [51], we introduced the basic idea of using formal methods for pre-injection analysis to derive a set of parameters to describe fault-injection scenarios. With the case study of clock synchronization [51], we highlighted the following key capabilities of our proposed pre-injection analysis approach: (a) support for traceability of fault-propagation over different functional block, (b) identification of a specific functional block which need to be further examined depending upon the inferences captured (via IT) and corresponding dependency list generated (via DT) for that block, and (c) support for modeling (or incorporating specification) of a specific functional block at a refined level of abstraction. Over this case study we also demonstrated the capabilities of IT/DT approaches to pinpoint a specific block (e.g., 2/3 Voter) which needed to be modeled to identify the cause of a failure (partial ordering problem of messages arriving at a specific node). In this case, 3827 tests were needed using classical FI versus 24 tests identified by our proposed approach. In both cases, the implementation had 3 fault cases and both techniques were correctly able to identify them. The identified parametric attributes include: round number, concurrency time-window, voting rate and numeric range for message sequences. It is to note that such information resulting over a pre-injection analysis facilitate (or guide) intelligent ways of determining influential (or key) variables to generate FI experiments for validating protocol operations.

In [49], [50], we demonstrated the effectiveness and efficiency of our approaches through the example of FT clock synchronization and FT Rate Monotonic Algorithm (FT-RMA) [25]. In the case of the fault-tolerant real-time task scheduling algorithm namely FT-RMA [49], [50], we were able to identify flaws in the analysis, and using IT/DT obtain the specific conditions to constitute effective FI test cases which, in fact, confirmed our identification of flaws. As a comparative analysis of our proposed pre-injection analysis technique with conventional approaches, we showed that even though FT-RMA protocols had gone through extensive simulation and random FI experiments, fault cases belonging to one of our derived equivalence classes of fault types were not identified. Typically, for simulations, task sets are randomly generated and due to their limitations in considering factors involving key schedulability criteria, these task sets would have a low probability to cover all key aspects of fault-tolerance and timing issues which we were able to capture during the formal treatment of pre-injection analysis of FT-RMA protocols.

Though the formal approach for analysis appears to be very attractive and effective, it has its own limitations. The foremost limitation is in the capabilities of formal techniques for representation of parametric attributes (e.g., specifying numeric bounds for variables, processor attributes, etc.), real-time deadlines, system workload conditions, etc. Furthermore, associated with these attributes, the corresponding formal verification process also needs to be developed. 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 acknowledge that further enhancement of the proposed pre-injection analysis is required to broaden the applicability of the approach.

Acknowledgments

We thank Peter Bokor and Marco Serafini for providing constructive feedback on the paper. Research supported in part by EC Indexsys, Inspire, TUD CASED and DFG GRK 1362 (TUD GKMM).

References

[1] G.A. Alvarez, F. Cristian, "Cesium: Testing Hard Real-Time and Dependability Properties of Distributed Protocols," *Proc. of IEEE WORDS'97*, pg. 2–8, 1997.

[2] T. Amnell, et al., "Uppaal – Now, Next and Future," *Modelling and verification of Parallel Processes*, LNCS – 2067, Springer-Verlag, pp. 100–125, 2001.

[3] J. Arlat, *et al.*, "Fault Injection for Dependability Validation : A Methodology and Some Applications," *IEEE Trans. Software Engineering*, SE 16(2), pp. 166–182, Feb. 1990.

[4] A. Arnold *et al.*, "An Experiment on the Validation of a Specification by Heterogeneous Formal Means: The Transit Node." *Proc. of DCCA-5*, pp. 24–34, 1995.

[5] D. Avresky, J. Arlat, J.-C. Laprie, Y. Crouzet, "Fault Injection for the Formal Testing of Fault Tolerance," *IEEE Trans. on Reliability*, vol. 45, pp. 443–455, 1996.

[6] S. Ayache, et al., "Formal Methods for the Validation of Fault-Tolerance in Autonomous Spacecraft," *IEEE FTCS-26*, 1996.

[7] E. Bayse, A. Cavalli, M. Nunez, F. Zaidi, "A Passive Testing Approach based on Invariants: Application to the WAP." *Computer Networks*, 48(2), pp. 247-266, June 2005.

[8] G. Bernot, "Software Testing Based on Formal Specifications," *Software Engg. Journal*, 6(6), pp. 387–405, Nov. 1991.

[9] S. Bishop, et al., "Rigorous Specification and Conformance Testing Techniques for Network Protocols, as Applied to TCP, UDP, and Sockets," *Proceedings of SIGCOMM 2005: ACM Conference on Computer Communications*, published as Vol. 35, No. 4 of Computer Communication Review, pp. 265–276, Aug. 2005.
[10] D.M. Blough, T. Torii, "Fault Injection Based Testing of Fault Tolerant Algorithms in Message Passing Parallel Computers," *Proc. of FTCS-27*, pp. 258–267, 1997.
[11] J. Boué, P. Pétillon, Y. Crouzet, "MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance," *Proc. of FTCS-28*, pp. 168–173, 1998.
[12] E. Brinksma, "Formal Methods for Conformance Testing: Theory Can be Practical," *CAV*, LNCS 1639, pp. 44–46, 1999.
[13] A. Brucker, B. Wolf, "Test-Sequence Generation with HOL-TestGen - With an Application to Firewall Testing," *Tests and Proofs*, LNCS 4454.Springer-Verlag, 2007.
[14] R. Butler, G. Finelli, "The Infeasibility of Quantifying the Reliability of Life Critical Real Time Software," *IEEE Trans. Software Engineering*, SE 19(1), pp. 3–12, Jan. 1993.
[15] J. Chang, D. J. Richardson, "Structural Specification-based Testing: Automated Support and Experimental Evaluation," *Proceedings FSE99*, pp. 285–302, Sept. 1999.
[16] W. Chen *et al.*, "Model Checking Large SW Specifications," *IEEE Trans. SE*, 7, pp. 498–520, July 1998.
[17] J. Christmansson, P. Santhaman, "Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms – Criteria for Error Selection Using Field Data on Software Faults," *Proc. of ISSRE*, pp. 175–184, 1996.
[18] S. Dawson, F. Jahanian, T. Mitton, T-L Tung, "Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection." *Proc. of FTCS–26*, pp. 404–414, 1996.
[19] K. Echtle, et al. "Evaluation of Deterministic Fault Injection for FT Protocol Testing," *Proc. of FTCS-21*, pp. 418–425, 1991.
[20] K. Echtle, M. Leu, "Test of FT Distributed Systems by Fault Injection," *FTPDS*, pp. 244–251, 1995.
[21] K. Echtle, M. Leu, "The EFA Fault Injector for Fault-Tolerant Distributed System Testing." *IEEE FTPDS*, pp. 28–35, 1992.
[22] A. Gargantini, C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," *Proc. of the 7th European Eng. Conf.* pp. 146–162. Springer-Verlag, 1999.
[23] M-C. Gaudel, "Testing Can be Formal Too?," *Proc. of TAPSOFT 95*, vol. 915, LNCS, pp. 82–96, May 1995.
[24] S. Ghosh, R. Melhem, D. Mossé, "Fault-Tolerant Rate Monotonic Scheduling." *Proc. of DCCA–6*, 1997.
[25] S. Ghosh, R. Melhem, D. Mossé, J.S. Sarma, "Fault-Tolerant Rate Monotonic Scheduling." *Real-Time Systems*,, vol. 15, no. 2, pp. 149–181, Sept. 1998.
[26] A. Goodloe, C. Pasareanu, D. Bushnell, P. Miner, "A Test Generation Framework for Distributed Fault-Tolerant Algorithms," *4th Workshop on Automated Formal Methods (AFM09)*, 2009.
[27] K.K. Goswami, R.K. Iyer, L. Young, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis," *IEEE Trans. on Computers*, 46(1), pp. 60–74, Jan. 1997.
[28] W. Gujjahr *et al.*, "Partition Testing vs. Random Testing: The Influence of Uncertainity," *IEEE Trans. on SE*, pp. 661–674, Sept/Oct. 1999.
[29] L. Heerink *et al.*, "Formal Test Automation: The Conf. Protocol with Phact," *Proc. of Test Conf.*, pp. 211–220, 2000.
[30] R. Iyer, D. Tang, "Experimental Analysis of Computer System Dependability," *Book chapter in 'Fault Tolerant Computer System Design'*, editor: D.K. Pradhan, Prentice Hall, pp. 282–392, 1996.
[31] M. Joseph, *Real-time Systems: Specification, Verification and Analysis*. Prentice Hall, London, 1996.
[32] J-C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology." *FTCS-15*, pp. 2–11, 1985.
[33] D. Larsson, R. Hahnle, "Symbolic Fault-Injection," *International Verification Workshop (Verify)*, vol. 259, pp. 85–103, 2007.
[34] J. Lehoczky, L. Sha, Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior." *Proceedings of IEEE RTSS*, pp. 166–171, December 1989.
[35] Yu Lei, D. Kung, Qizhi Ye, "A blocking-based approach to protocol validation." *Computer Software and Applications Conference, COMPSAC 2005*, pp. 301–306, July 2005.
[36] R.Lent, "A testbed validation tool for MANET implementations." *MASCOTS 2005*, pp. 381–388, Sept. 2005.
[37] C.L. Liu, J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." *Journal of the ACM*, 20(1), pp. 46–61, January 1973.
[38] S. Mullender (Ed.), *Distributed Systems*, Addision-Wesley, 1993.
[39] V. Okun, P.E. Black, Y. Yesha, "Testing with Model Checker: Insuring Fault Visibility." *WSEAS Transactions*, 2003.
[40] S. Owre, J. Rushby, N. Shankar, F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Trans. Software Engineering*, SE 21(2), pp. 107–125, February 1995.
[41] S. Owre, "Random Testing in PVS," In *Workshop on Automated Formal Methods*, 2006.
[42] M. Pandya, M. Malek,"Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks," *IEEE Trans. on Computers*, 47(10), pp. 1102–1112, Oct. 1998.
[43] C. Pasareanu, et al., "Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing NASA Software," *Proc. of Int. Symp. on SW Testing and Analysis*, pp. 15–26. ACM Press, 2008.
[44] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, R. Iyer, "SymPLFIED: Symbolic Program-Level Fault-Injection and Error-Detection Framework," *IEEE DSN*, June 2008.
[45] S. Rajan, N. Shankar, M.K. Srivas, "An Integration of Model-Checking with Automated Proof Checking," *Computer-Aided Verification, CAV '95*, LNCS 939, pp. 84–97, 1995.
[46] J. Rushby, "Formal Methods and the Certification of Critical Systems," *SRI-TR CSL-93-7*, Dec. 1993.
[47] J. Rushby, F. von Henke, "Formal Verification of Algorithms for Critical Systems." *IEEE Trans. on SE*, 19(1), pp. 13–23, 1993.
[48] M. Singhal, N.G. Shivaratri, *Advanced Concepts in Operating Systems*, McGraw Hill, 1994.
[49] P. Sinha, N. Suri, "Identification of Test Cases Using a Formal Approach," *Proc. of FTCS–29*, pp. 314–321, 1999.
[50] P. Sinha, N. Suri, " On the Use of Formal Techniques for Analyzing Dependable RT Protocols," *Proc. of RTSS*, pp. 126–135, Dec. 1999.
[51] N. Suri, P. Sinha, "On the Use of Formal Techniques for Validation." *Proc. of FTCS-28*, pp. 390–399, 1998.
[52] T. Suzuki et al., "Murate: A Protocol Modeling & Verification Approach Based on a Specification language and Petri Nets," *IEEE Trans. on SE*, SE 16, pp. 523–536, May 1990.
[53] S. Tao, et al., "Focused Fault Injection of Software Implemented Fault Tolerance Mechanisms of Voltan TMR Nodes." *Distributed Systems Engineering*, 2(1), pp. 39–49, March 1995.
[54] T. Tsai, S.J. Upadhaya, H. Zhao, M.-C. Hsueh, R.K. Iyer, "Path-Based Fault Injection," *Proc. 3rd ISSAT Conf. on R&Q in Design*, pp. 121–125, 1997.
[55] J. Tretmans, "Specification Based testing with Formal Methods: From Theory via Tools to Applications," *Proc. of FORTE 2000*.
[56] N. Varma, et al., "CAGILY: An Approach for Developing Test Suites for Component-Based Systems," *IASTED SEA*, Nov. 2003.
[57] J. Voas, G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*, John Wiley & Sons Ltd, New York, 1998
[58] C.J. Walter, P. Lincoln, N. Suri, "Formally Verified On-Line Diagnosis" *IEEE Trans. on Software Engg.*, Nov. 1997.
[59] W. Wang, et al., "The Impact of Fault Expansion on the Interval Estimate for Fault Detection Coverage," *Proc. of FTCS–24*, pp. 330–337, 1994.
[60] Shu Xiao, et al., "Integrated TCP/IP Protocol Software Testing for Vulnerability Detection." *ICCNMC*, Oct. 2003.

Neeraj Suri holds the TUD Chair Professorship at the Dept of CS, TU Darmstadt Germany. Details on his professional activities can be found at http://www.deeds.informatik.tu-darmstadt.de/.

Purnendu Sinha is a Staff Researcher at General Motors R&D, Bangalore, India. His research interests include dependable distributed systems, software engineering, and formal-methods based V&V.

APPENDIX: PVS SPECIFICATION OF PP (WLS) ALGORITHM [58]

```
pp[m: posnat, n: posnat, T: TYPE, error: T, BAD: {x: T | ¬ x = error}, GOOD:
    {x: T | (¬ x = error) ∧ (¬ x = BAD)}, Val:
    [upto[m] → {x: T | ¬ (x = error ∨ x = BAD ∨ x = GOOD)}]]: THEORY
 BEGIN
  rounds: TYPE = upto[m]
  t: VAR T
  fcu: TYPE = below[n]
  fcuset: TYPE = setof[fcu]
  fcuvector: TYPE = [fcu → T]
  G, p, q, z: VAR fcu
  v, v1, v2: VAR fcuvector
  caucus: VAR fcuset
  r, R, R2: VAR rounds
  PSET: TYPE = fcu
  pset: VAR setof[PSET]
  i, j, k, i2, j2: VAR PSET
  Accuse, OldAccuse: VAR [PSET, PSET → bool]
  AllDeclare: VAR [PSET, PSET → bool]
  IMPORTING card_set[fcu, n, identity[fcu]], finite_cardinality[fcu, n, identity[fcu]], filters[fcu], hybridmjrty[T, n, error]
  statuses: TYPE = {symmetric, manifest, good}
  status: [fcu → statuses]
  g(z): bool = good?(status(z))
  s(z): bool = symmetric?(status(z))
  c(z): bool = manifest?(status(z))
  cs(caucus): fcuset = filter(caucus, c)
  ss(caucus): fcuset = filter(caucus, s)
  gs(caucus): fcuset = filter(caucus, g)
  fincard_all: LEMMA fincard(caucus) = fincard(cs(caucus)) + fincard(ss(caucus)) + fincard(gs(caucus))
  send: [T, fcu, fcu → T]
  send1: AXIOM g(p) ⊃ send(t, p, q) = t
  send2: AXIOM c(p) ⊃ send(t, p, q) = error
  send4: AXIOM s(p) ⊃ send(t, p, q) = send(t, p, z)
  send5: LEMMA send(t, p, q) = send(t, p, z)

  HybridMajority(caucus, v): T = PROJ_1(Hybrid_mjrty(caucus, v, n))

  HybridMajority1: LEMMA
    fincard(gs(caucus)) > fincard(ss(caucus)) ∧ (∀ p: g(p) ∧ (p ∈ caucus) ⊃ v(p) = t) ∧
      t ≠ error ∧ (∀ p: c(p) ∧ (p ∈ caucus) ⊃ v(p) = error) ⊃ HybridMajority(caucus, v) = t

  HybridMajority2: LEMMA
    (∀ p: (p ∈ caucus) ⊃ v1(p) = v2(p)) ⊃ HybridMajority(caucus, v1) = HybridMajority(caucus, v2)

  HybridMajority3: LEMMA
    HybridMajority(caucus, v) = t ∧
      (∀ p, q: g(p) ∧ g(q) ∧ (p ∈ caucus) ∧ (q ∈ caucus) ⊃ (v(p) = v(q) ∧ v(p) ≠ error)) ∧
       fincard(gs(caucus)) > fincard(ss(caucus)) ∧ (∀ p: c(p) ∧ (p ∈ caucus) ⊃ v(p) = error)
      ⊃ (∀ p: g(p) ∧ (p ∈ caucus) ⊃ v(p) = t
  Syndrome(R, j, i, OldAccuse): T =
      IF OldAccuse(i, j) ∨ (¬ Val(R) = send(Val(R), j, i)) THEN BAD
      ELSE GOOD ENDIF

  KDeclareJ(pset, R, OldAccuse, j, k): bool =
      HybridMajority(pset, λ i: send(Syndrome(R, j, i, OldAccuse), i, k)) = BAD

  PP(pset, R, OldAccuse)(i, j): RECURSIVE bool =
    IF R = 0 THEN FALSE
    ELSE KDeclareJ(pset, R, OldAccuse, j, i) ∨
           PP(pset, R − 1, (λ i2, k: OldAccuse(i2, k) ∨
                    (∃ j2 (KDeclareJ(pset, R, OldAccuse, j2, i2) ≠
                         (send(Syndrome(R, j2, k, OldAccuse), k, i2) = BAD))))) (i, j)
    ENDIF MEASURE (λ pset, R, OldAccuse: R)

  Soundness_Prop(R): bool =
      (∀ i, j, pset, OldAccuse: g(i) ∧ (i ∈ pset) ∧ (j ∈ pset) ∧
           fincard(gs(pset)) > fincard(ss(pset)) + 1 ∧
            PP(pset, R, OldAccuse)(i, j) ∧
             (∀ p, q, k: ((g(p) ∧ g(q) ∧ OldAccuse(p, k)) ⊃ OldAccuse(q, k) ∧ (c(k) ∨ s(k))))
         ⊃ c(j) ∨ s(j))

  Soundness: LEMMA Soundness_Prop(R)

  Completeness_Prop(R): bool =
      (∀ i, j, pset, OldAccuse: g(i) ∧ (i ∈ pset) ∧ (j ∈ pset) ∧
           (c(j) ∨ (s(j) ∧ (∀ t, p: send(t, j, p) ≠ t))) ∧ fincard(gs(pset)) > fincard(ss(pset)) + 1
         ⊃ PP(pset, R, OldAccuse)(i, j))
  Completeness: LEMMA (∀ R: Completeness_Prop(R) ∨ R = 0)

  Empty(i, j): bool = FALSE

  Final_Soundness: THEOREM
    (∀ i, j: g(i) ∧ fincard(gs(fullset[fcu])) > fincard(ss(fullset[fcu])) + 1 ∧
          PP(fullset[fcu], R, Empty)(i, j) ⊃ c(j) ∨ s(j))

  Final_Completeness: THEOREM
    (∀ i, j: g(i) ∧ (c(j) ∨ (s(j) ∧ (∀ t, p: send(t, j, p) ≠ t))) ∧
          fincard(gs(fullset[fcu])) > fincard(ss(fullset[fcu])) + 1 ∧ R > 0
        ⊃ PP(fullset[fcu], R, Empty)(i, j))
 END pp
```