# Finding Almost-Invariants in Distributed Systems

Maysam Yabandeh[‡], Abhishek Anand[†], Marco Canini[*], and Dejan Kostić[*]
[*]EPFL, Lausanne, Switzerland
[†]Cornell University, Ithaca NY, USA, (work done during an internship at EPFL)
[‡]Yahoo! Research Barcelona, Spain (work done during Ph.D. studies at EPFL)
maysam@yahoo-inc.com abhishek.anand.iitg@gmail.com {Marco.Canini,Dejan.Kostic}@epfl.ch

***Abstract*—It is notoriously hard to develop dependable distributed systems. This is partly due to the difficulties in foreseeing various corner cases and failure scenarios while implementing a system that will be deployed over an asynchronous network. In contrast, reasoning about the desired distributed system behavior and the corresponding invariants is easier than reasoning about the code itself. Further, the invariants can be used for testing, theorem proving, and runtime enforcement.**

**In this paper, we propose an approach to observe the system behavior and automatically infer invariants which reveal implementation bugs. Using our tool, Avenger, we automatically generate a large number of potentially relevant properties, check them within the time and spatial domains using traces of system executions, and filter out all but a few properties before reporting them to the developer. Our key insight in filtering is that a good candidate for an invariant is the one that holds in all but a few cases, i.e., an "almost-invariant". Our experimental results with the XORP BGP implementation demonstrate Avenger's ability to identify the almost-invariants that lead the developer to programming errors.**

## I. Introduction

Implementing and deploying highly dependable distributed systems is difficult for a number of reasons, including the sheer system size, concurrency issues, the number of unforeseen events, and the difficulty in structuring protocols that run over asynchronous networks.

The approaches for making distributed systems more reliable have evolved from debugging using log inspection to more complex techniques such as property checking [16], [18], [22], model checking [14], [22], and enforcing the invariants at runtime [22]. The latter approaches require the developer to specify the desired system behavior in the form of invariants that are supposed to hold at all times. Although reasoning about invariants is arguably easier than reasoning about the source code itself, the developer is still expected to provide the invariants. This task becomes more and more difficult as the system gets larger and more complicated, and as the developer starts dealing with various corner cases. For example, in distributed systems in which various network failures can occur, reasoning about an invariant that holds under all failure conditions can be difficult. While some distributed systems have been written with invariants in mind [15], many have not. Although others have shown that it is possible to discover invariants [7], [8] and even specifications [2], [5] of single-machine code, invariant inference is still an open and important challenge in distributed system implementations.

We observe that, due to the difficulty of dealing with various issues in the deployment environment, there exist a potentially large number of important distributed system invariants that only get violated under certain conditions (and would be discarded if the existing tools for single-machine code [7], [8] were to be applied). We refer to such properties as *"almost-invariants"*[1] . Most often, an almost-invariant gets violated due to a rare manifestation of a bug that needs to be fixed.

In this paper, we introduce a new tool, Avenger, for inferring almost-invariants in distributed system implementations. Our approach leverages the rarity of the manifestation of inconsistencies and emergent behaviors in complex distributed systems and looks for the almost-invariants across long and varied distributed system executions. Our tool, Avenger, uses a grammar and developer input to generate a large number of potential properties. The properties are evaluated for validity both over time and *spatially*, e.g., across the system state.

Our key insight is in realizing that the complexity of distributed environment (with its failure modes and the underlying asynchronous network) makes it hard to reach bulletproof distributed system implementations. The last 1% bugs manifest very rarely which makes them even harder to detect. For example, a bug can be introduced when the developer fails to take a particular sequence of inputs, events, or failures (corner case) into consideration when writing the program, and as a consequence the system does not behave as the developer intended. It is also possible that some emergent behavior could be the cause for less than perfect system operation. Using these insights, Avenger ultimately reports a handful number of almost-invariants that hold in most of (but not all) the cases.

We make the following contributions:

- We introduce a new automatic testing technique for distributed system implementations based on identifying the almost-invariants in execution traces. The inferred almost-invariants are those that are likely to point to programming errors.
- We demonstrate that our tool handles the peculiarities of distributed invariants that span state across multiple nodes, as well as invariants in the spatial domain (where only some machines in the system are violating a property at a given point in time).
- We demonstrate that our tool exposed a problem in handling 4-byte AS numbers in the BGP protocol implemented in C++ within the XORP open-source router [11].

[1]A property expresses a relation between some variables (including iterator variables) which can be evaluated as true or false. Throughout this paper, we use the term *invariant* to refer to a property that is never violated.

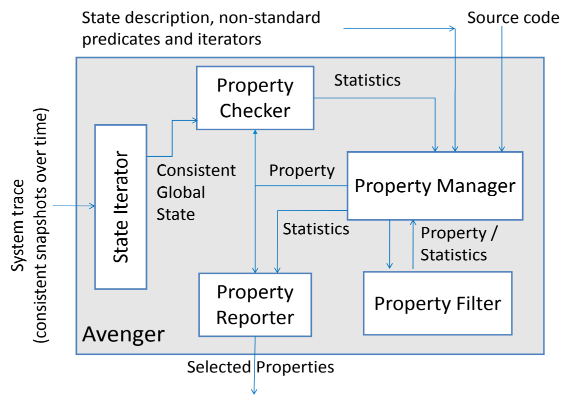

Fig. 1: High-level overview of Avenger

## II. Avenger

Avenger is a centralized, off-line tool that the programmer runs either prior to, or after deployment. At a high-level, Avenger works by examining consistent snapshots of state across multiple nodes. The programmer prepares her distributed system for analysis by Avenger in three steps: i) providing the annotations, ii) compiling the executable, and iii) generating the state traces. First, the programmer supplies a C++ header file describing a node object, which contains the state variables of interest. To drive the generation of potential properties, the programmer also supplies two C++ files that contain predicates and iterators that are not already supplied by Avenger. The final piece of developer input is the values of configuration parameters that control the type and the number of potential properties that will be generated. Second, to produce the Avenger executable, the programmer compiles and links: 1) the programmer-supplied files, 2) any application libraries that are needed to access state or invoke functions, and 3) the Avenger library. Finally, the programmer supplies state traces to drive the Avenger executable. Entries in the trace are system snapshots recorded over time: each snapshot is a consistent set of nodes' states.

Figure 1 provides a high-level overview of Avenger by depicting the key components and the type of data flowing between them. The *Property Manager* uses a system specification to produce a pool of potential properties. The *State Iterator* examines the system states (sequentially or in parallel) and feeds the consistent, global system state to the *Property Checker*. The property checker checks the properties against each state, updates their statistics and invokes the *Property Filter* at coarse timescales. Using the property statistics, the filter attempts to reduce the size of the property pool in an effort to speed up overall execution time. At the end of the state iteration, the *Property Reporter* module selects a handful of the almost-invariants in the property pool and reports them to the developer for further inspection.

Avenger is primarily a testing tool. To pinpoint a programming error or discard a reported property altogether, the programmer relies on available domain knowledge and source code inspection. She can also feed the almost-invariant to a debugging tool that produces a sequence of steps that lead to an invariant violation, e.g., [16], [22]. Avenger is best-suited for finding almost-invariants via violations of safety properties (those that should hold at all times). It is not capable of dealing with liveness properties, i.e., those that hold eventually.

### A. Using Avenger

*1) Developer's input to property generation:* Avenger generates properties referring to the variables that represent the state of system nodes. Some development frameworks make the state of a node explicit [13], which in turn makes it possible for Avenger to automatically extract the relevant variables. In the general case of arbitrary C++ code, Avenger uses a developer-provided system specification to become aware of the relevant pieces of state.

A system specification consists of two parts: Variables and Predicates. We use examples from our application of Avenger to XORP [11], a popular open-source routing platform, to illustrate Avenger usage. Providing this input is straightforward: one of our researchers unfamiliar with XORP required less than a day to produce the required specification.

**Variables** The state-related input to Avenger is a file called Variables.h that contains the specification of the structure of a system node state. Avenger provides a simple syntax to express the state structure: the user simply needs to write the identifiers of the state variables and their corresponding data types. An identifier is basically a variable name. Global functions or public class methods with empty parameter lists can also be used as identifiers provided that calling these functions will not change the state. In these cases the identifier is the function name followed by the symbol `()`. A reference to a basic data type is expressed with the `//VARIABLE` keyword, while a container data type is referenced with the `//CONTAINER` keyword. A XORP example is shown below:

```
//CONTAINER IPv4 { uint32_t addr() };
//CONTAINER OriginTable<IPv4>  { Trie<IPv4,
   IPv4RouteEntry*> route_container() };
```

Avenger supports all data types in C++ ranging from built-in data types such as $integers$, $floats$, and $booleans$, to complex user-defined classes and structs. Polymorphic types and templates are also supported.

Further, the syntax allows a few annotations that appears in between the `/*` and `*/` symbols, e.g., `/*iterator*/`, informs Avenger whether a certain data type is iterable. Automatically, Avenger supports standard container data types such as STL array, vector, set, map, etc. Extending Avenger to new iterable data types is straightforward and just requires an implementation of the Iterator interface provided by Avenger and a simple annotation in Variables.h.

**Predicates** Similar to other invariant inference tools [7], Avenger provides predicates that correspond to standard operators (equality, membership, etc.). In addition, Avenger is extensible by allowing the user to specify additional predicates of interest. The user does this by populating the Predicates.h file with *predicate templates* and providing the corresponding logic in Predicates.cc.

In general, a predicate combines the variables using operators, e.g., $parent \in children$. A predicate template expresses an operation over particular data types. Later during the evaluation of properties, Avenger calls the implementation of the predicate template in the Predicates.cc file. For example, the predicate template we added in XORP to access a non-standard data structure (details are in Section III) is as follows:

```
bool predicateCanReach(
     const OriginTable<IPv4> &rt,
     const IPNet<IPv4> &net)
{
    const IPv4RouteEntry* re =
       rt.lookup_route(net);
    return (re != NULL && re->vif() != NULL);
}
```

*2) Trace Collection and State Iteration:* Avenger assumes that an external module generates a trace of globally consistent system states. These states can be obtained by periodically recording the global state of a live execution, similarly as in WiDS [16]. Alternatively, a module could iterate over the traces of distributed system events, create the global state after each step [10] (e.g., handler execution on a node), and feed it to the state iterator. The traces of the system events can be obtained from the log files recorded during live deployments.

The consistent snapshots are fed to Avenger as sets of objects that describe node state (recall that the Variables.h file describes the relevant state). This means that Avenger can be applied to systems written in a variety of programming languages, provided that these systems collect consistent snapshots and convert them to C++ objects to present them to Avenger. Calling existing functions on the state variables might require linking Avenger with parts of the application code that defines the data structures.

## B. Avenger Design and Implementation

*1) Property Manager:* Avenger's Property Manager uses a developer-provided specification of the system to generate the set of potential properties. The main challenge here is to support the generation of all the relevant properties that can hold in the distributed system under scrutiny. Next, we describe our approach to meeting this challenge.

Properties are expressed internally using a grammar similar to first-order logic. The grammar is sufficiently *expressive* to derive complex system properties which, for example, iterate over container data types while tying together states from different nodes, as later shown in Section III. Fundamentally, each property is a disjunction of *predicates* or their negation. As in first-order logic, a predicate represents a relation between variables and can be evaluated as true or false. For example, $n_1.hashId \in n_0.neighbor\_list$, is a predicate which is evaluated to true if the node $n_1$'s hash ID is a member of the node $n_0$'s neighbor list. A predicate consists of an operator, e.g., the membership operator ($\in$), and variables, such as the hash ID and the list of neighbors in the above example. To help Avenger discover properties with universal quantification, the user expresses iterators that show Avenger how to identify all the members of the quantification universe. At the top level, a property iterates over the set of nodes in the distributed system, potentially with multiple nested iterations. Further down in the nested loops, a property may iterate over arbitrary container-type variables, such as arrays, vectors, lists, sets, and maps.

Formally, the following describes the property syntax:

$$\begin{aligned} Pr_i: \quad & \forall n_0 \in nodes \ldots \forall n_m \in nodes: \\ & \forall s_0 \in n_q.v_0 \ldots \forall s_k \in n_{q'}.v_k: \\ & l_0 \vee l_1 \cdots \vee l_t \end{aligned}$$

which selects $m+1$ nodes, $n_i$ ($0 \leq i \leq m$), and $k+1$ variables, $v_i$ ($0 \leq i \leq k$), as the universes of quantifications, and calculates the disjunction of $t+1$ literals, $l_i$ ($0 \leq i \leq t$). Each literal represents a predicate $P$ or its negation $\neg P$. The complete set of properties that can be generated is prohibitively large. For example, $n$ binary literals can be combined to form $2^{2^n}$ distinct properties (formulae)[2].

In an effort to reduce computational cost, we do not consider conjunction of predicates[3]. This decision has limited impact on the Avenger's ability to identify important almost-invariants, as every first-order logic formula $F$ can be converted to a conjunction of some other first-order logic formulas $F_{cf}$, where $F_{cf}$ does not include any conjunction. While evaluating the properties over a trace, Avenger tracks their *holding rate*, i.e., the cumulative rate of evaluating to true. In the end, Avenger selects the top almost-invariants to report to the user (more details are in Section II-E). Because the rank on the report list of each conjunction-free formula, $F_{cf}$, would be higher than the whole formula, $F_w$, formula $F_w$ will not be selected in the process anyway.

*a) Property Generation:* The following presents a high-level description of how Avenger generates the initial properties. First, Avenger parses the list of variables and recursively expands the container data types to produce a tree of variables and iterators, each of which is annotated with its type and sub type. Cyclic dependencies can cause the recursive expansion to continue indefinitely. Although our state descriptions had no cyclic dependencies, in future we can limit the tree depth to avoid this potential problem. The role of this tree structure is explained later. Avenger then reads in the predicate templates and generates a first set of predicates by instantiating predicate templates with all the possible combinations of variables with the appropriate data types. These generated literals make use of a single system node reference, that is, any combination of variables is scoped within the state of a single node (e.g., `contains(n.neighbor_list, n.hashId)`). During a second iteration, Avenger increases the number of system node references and instantiates a second set of literals with all the possible and appropriate combinations of variables whose scope is now enlarged to address the state of two system node states (e.g., `contains(n0.neighbor_list, n1.hashId)`, `contains(n1.neighbor_list, n0.hashId)`). This procedure is repeated until the number of node references reaches the maximum allowed, set by the `MAX_NF` control parameter. And so, one set of predicates is created for each number of node references. Note that the number of generated

[2]Observe that $n$ boolean literals can have $2^n$ unique configurations and a property can be specified uniquely by the set of configurations for which it is true. We can have $2^{2^n}$ subsets of $2^n$ configurations.

[3]If the programmer so desires, she can include a conjunction in a custom predicate implementation.

predicates increases polynomially w.r.t. MAX_NF with degree equal to the arity of the predicates. For example, for binary predicates, the number of predicates increases quadratically with MAX_NF.

Finally, Avenger iterates over each set of predicates and generates the properties by combining through disjunction all the possible combinations of predicates (and their negation) taken one at a time, then two at a time and so on. The maximum number of predicates in a single property is controlled by a parameter MAX_NT. The number of properties with $n$ literals is of the order $O(n^{MAX\_NT})$. Rather than leaving the user to guess a suitable value of MAX_NT, Avenger provides her the option of specifying the maximum number of properties that she wants to generate in each class.

Lastly, Avenger uses the tree of variables and iterators to produce a valid C++ qualifier for each state variable which is plugged into the properties' internal representation format in order to produce a series of C++ statements each of which implements the evaluation of a predicate.

## C. Property Checker

The main task of the Property Checker is to update property statistics that enable filtering and reporting later on. This component is invoked for each global state snapshot. Upon each invocation, the Property Checker evaluates every property (which yields a boolean value) and assigns an individual property *score*. Finally, it adds the score to the cumulative holding rate which is kept for each property in the pool. A simple property accounting strategy is to score a property with one if the property holds at all nodes in the snapshot, and zero otherwise. The holding rate statistic is later used by the property filter module, as well as during the final ranking of the properties, with a general goal of ranking higher the properties that end up violated in fewer instances.

**Spatial Accounting** One major difference between checking the properties in distributed systems vs. single-machine is that some properties might hold at a subset of nodes but not for all of them. Using the example property $\forall n \in nodes : P_1$, the predicate $P_1$ can evaluate to true for only some of the nodes in the global snapshot. We handle cases like this using *spatial accounting* which computes a fractional score: we set the score to the number of times that the property holds divided by the total number of samples (the number of combinations of nodes on which the property is evaluated in a snapshot).

**Fast Accounting** As the property detection process is likely to start with a large number of possible properties, checking the validity of all properties has to be done *quickly and efficiently*. To efficiently evaluate the properties we observe that many properties share the iteration over the same iterable container variables because far more properties are generated than iterable types exist. The idea is to iterate over all the iterable containers only once and re-evaluate those predicates/properties that depend on a particular iterator only when such iterator changes. However, this needs to be done correctly while handling iterator dependencies and nestings.

**Scalability** The complexity of checking a single snapshot in the trace is $\Omega((n)^{MAX\_NF})$, where $n$ is the number of nodes. This is because there are $MAX\_NF$ universal quantifications over nodes and hence a nested loop with nesting depth of $MAX\_NF$ is required. Our most common setup has $MAX\_NF = 2$. The complexity further depends upon the size of the iterable sets in a node's state. Let us assume that there are $k$ iterable sets in a node's state where size of $i^{th}$ set is $O(S_i(n))$. Then, time complexity of evaluation can be stated as $O(n * \prod_{i=1}^{k} S_i(n))^{MAX\_NF}$, as each of these iterable set will have its own nested loop running over its elements. Note that the total number of properties does not depend on the number of nodes and in each iteration, only a subset of literals which depend on the modified iterators are evaluated. For simplicity we consider the amount of work done in each iteration as constant in this analysis. Moreover, each snapshot is evaluated independently of others. So, the time complexity increases linearly with the number of snapshots if we assume that the sizes of iterable sets remain the same.

**Load Balancing** Given the ubiquity of cheap clusters of multi-core machines, an important challenge lies in harnessing the available computational power to speed up checking and, ultimately, the entire property inference process. We have parallelized the property checking task by assigning to each CPU core in the cluster the responsibility of a disjoint subset of snapshots. For this purpose, we use the operation modulo $N$ over $S$, where $N$ is the number of checking processes and $S$ is the explored state index, so that process $x$ checks the snapshots for which $S \mod N = x$. We find this simple load-balancing technique to work well in practice. In our evaluation, 10 quad-core machines were sufficient to bring Avenger execution time to less than a few hours for even the longest state traces we encountered (860,000 states) [21].

## D. Property Filter

Recall that the properties in our interest are the ones which have a holding rate very close to 100%. The properties which have held much less are unlikely to be chosen at the end of the property inference process. When the property filter is invoked, we remove the properties that hold less than a threshold. Overall, there is a trade-off between the tool accuracy and the speed of execution which can be controlled by the threshold filtering value, FILTER_HOLD_RATIO (default value is 0.95). At the extreme, one could complete the run of the tool while keeping all the properties.

## E. Property Reporter

The set of reported properties should ideally reveal all the manifested bugs in the traces of the system run. This set should not be too large, as inspecting too many properties can overwhelm the developer. Accordingly, the falsely reported properties waste developer's time due to the required efforts to reject them. Thus, one of our design goals is to report only a small number of properties. The property reporter module accomplishes this task, and its REPORT_SIZE parameter can be used to change the number of reported properties (in our experiments, we set this parameter to 10).

**Prioritizing properties** Given our observation that the bug-manifesting input sequences are usually very rare, we look for properties that hold in most of, but not all, cases. Thus, we sort the potential almost-invariants by their holding rate in descending order. The property reporter then iterates over the candidate list of almost-invariants and applies the following.

**Eliminating equivalent properties** An important step for improving the quality of the reported almost-invariants is to reduce the number of equivalent properties by reporting only one property from each *equivalence set*. The properties with the same holding ratio (to a high precision) are likely to be equivalent. We use this simple statistic apart from mathematical equivalence because there could be two properties that are not mathematically equivalent, but are in practice checking the same system aspect. We therefore put the almost-invariants with the same holding rate in the same equivalence set and output only one almost-invariant chosen at random. The original equivalence set can still be reported.

**Simplifying properties** Finally, we simplify the properties to help reduce the human effort in analyzing the almost-invariants. Recall that each property $Pr$ is a disjunction of some predicates and thus can be split into two or more properties $Pr_1 \vee Pr_2 \vee \ldots \vee Pr_n$ where each simpler property $Pr_i$ contains a disjoint subset of the predicates in $Pr$. Because of what we refer to as the *Or Effect*, we need to make sure that $Pr$ does not subsume any simpler property that is in the candidate list. To address this problem, we use a simple independence statistical test.

## III. Evaluation

In this section, we report our results of using Avenger to analyze the BGP implementation in XORP version 1.6[4]. BGP is the standard inter-domain routing protocol in the Internet.

**Implementation Details** The changes we made to the XORP source code were minimal, took about two weeks to perform (we had no previous experience with the XORP platform), and consisted of: 1) incorporating an existing snapshot algorithm and 2) writing C++ serialization and deserialization routines for the Routing Information Base (RIB). The variables we included in the description file simply reflect the state of the XORP RIB process: the routing tables and their route entries, each made of a network prefix, a metric, a next hop, a pointer to an egress interface, and policy flags. We defined equality predicates for simple data types (e.g., IP addresses, route prefixes) and applied basic knowledge of routing domain to define the reachability predicate that checks the membership of a route prefix in a routing table (essentially access to a non-standard data structure, Section II-A1).

**Experimental Setup** Our testbed makes use of virtual interfaces to enable multiple XORP instances to communicate over a synthetic topology that is installed within a single 48-core machine. This relatively simple setup does not delay or drop packets, but it allows us to bring interfaces down and up to simulate link and node failures as to expose BGP behavior. In our experiments, we configure multiple BGP instances in clique (full-mesh) and b-clique topologies [17] with 4, 8, 16 and 32 nodes. Inspired by the BGP problems reported in [1], we mixed 16-bit and 32-bit AS numbers in the attempt to uncover unexpected BGP behaviors in our testbed. Finally, we drove the experiments with a script that can advertise and withdraw arbitrary prefixes, and trigger link and node failures. We conservatively marked snapshots as steady state if there were no UPDATE BGP messages in the past minimum route advertisement period (30 s by default). In total, there were 11,885 snapshots that were fed to our tool. Starting from the small set of RIB variables and added predicates, Avenger generated 3,700 properties and identified two relevant almost-invariants (only one shown in discussed in this section for lack of space), out of a total of four reported (10 requested).

**Property BGP1: Universal reachability.** This almost-invariant covers complete reachability which is one of the most important features of the Internet. This invariant is close in spirit to path visibility, an important BGP property defined in [9], which specifies that every router that has a usable path to a destination learns at least one valid route to that destination. The property Avenger generated is as follows (after very little cleanup):

$\forall n_1 : \forall n_2 : \forall r \in n_2.ebgpTable : r \in n_1.ebgpTable$

Surprisingly, in our experiments we found the property being violated a few times during steady state. After careful investigation, we found that the property was systematically violated when two or more 32-bit AS BGP speakers were in the topology and at least one of these speakers was advertising one or more owned prefixes.

This almost-invariant is evidence that Avenger has all the required expressiveness to produce two iterations over the nodes and one iteration over a custom container type with small amount of developer input (to access a non-standard data structure). In addition, this almost-invariant ties together state from multiple nodes in a distributed system implementation.

**Discussion** Given the difficulty of building reliable distributed systems, a tool that helps to identify hard-to-find programming errors is useful. Avenger was effective for every system we examined. Moreover, two out of four reported invariants turned out to be relevant in the case of XORP, with one leading to a previously unknown problem. We note that the seemingly high false positive rate is typical in anomaly-finding tools [3].

Determining the relevance of the reported properties took less than a day and primarily involved checking the source code and, when necessary, the logs and consistent snapshots. One case required a change in the experimental setup. The elimination of properties was quick, as the "useless" properties turned out to be: 1) semantic equivalents of other properties (e.g., referencing an internal subfield of a routing entry that is as unique as the entry value itself), or 2) a result of the expected system operation (e.g., the entries in the routing tables of two nodes have different next hops). Although the inspection process was fairly straightforward, domain knowledge was still required during source code and property inspection.

Avenger now effectively shifts the burden on the generation of system traces. These traces should document the operation of the distributed system under a variety of expected workloads. Perhaps more importantly, the distributed system

[4]Due to lack of space, results of applying Avenger to the RandTree overlay tree, the Chord distributed hash table, and the Paxos consensus protocol (all within the Mace [13] distributed system framework) are described in the accompanying technical report [21].

should be subjected to complex failure scenarios (faultloads). However, achieving the desired variety, especially in the case of failures, might require substantial resources and execution time. We note that the recent availability of model checking tools for distributed system implementations [14] offers the possibility of quickly obtaining a large number of long executions that explore many possible interleaving of the node actions and failures in so-called *random walks*.

## IV. Related Work

**Using invariants during development and deployment** Some distributed systems and algorithms have been designed from ground up with the invariants in mind. The best example is perhaps Paxos [15], a distributed algorithm for achieving consensus over asynchronous networks.

Killian *et al.* [14] have manually specified safety and liveness properties in their MaceMC model checker for distributed system implementations, and used them successfully to identify bugs in several systems. WiDS [16] similarly looks for violations of known invariants.

Avenger goes in the opposite direction in the sense that by checking the traces of the system execution, it detects the inconsistencies and proposes the invariants corresponding to the observed inconsistencies.

**Invariant inference for single-machine code** Static analysis [4], which involves source code examination without execution, can be used to infer properties. This approach is typically sound, but, due to some issues (e.g., pointers), static analysis is in practice too conservative.

Dynamic approaches overcome the shortcomings of static analysis by i) generating a large number of possible properties, and ii) relying on test cases to exercise the code behavior. Daikon [7], [8] filters out any automatically generated property that is violated during a test run, which assumes that the code or specification that is used to generate the properties is correct. This assumption does not always hold.

**Self-consistency** Our work is somewhat related in spirit to checking the self-consistency of source code [6]. This work uses manually-defined templates and performs static analysis of single-machine source code to detect common patterns in sequences of commands. It then reports the deviations from these patterns as inconsistencies to the developer. However, it cannot deal with the bugs that are not a deviation from the similar programming patterns in the rest of the code. DIDUCE [12] is somewhat similar in spirit to Avenger and the work in [6], but it works on single-machine Java code and has significant execution overhead.

**Invariant inference for distributed algorithms** The work on automatic detection of properties in SPIN [19] can verify whether two variables are related by basic operators. The work that is perhaps closest to ours is [20], whose goal is to infer the safety properties of distributed algorithms using Daikon. This work simulates the execution of multiple IO automata that contain a specification of a distributed algorithm in an abstract form. In contrast, our work produces properties which correspond to inconsistencies in distributed system implementations, and deals with properties in the spatial domain.

## V. Conclusions

In this paper, we tackle the problem of automatically inferring distributed system properties. Our tool, Avenger: i) generates a large number of potential properties, ii) checks them within the time and spatial domains using traces of system behavior, and iii) chooses only several *almost-invariants* that warrant inspection by the programmer. Avenger increases the resilience of distributed systems as inspection of these properties can uncover programming errors.

## Acknowledgments

We are grateful to Simon Schubert and Jiaqing Du for their help with the XORP testbed. This project is supported by the Swiss NSF (grant FNS 200021-125140). Marco Canini was funded in part by a grant #2103 from the Hasler foundation.

## References

[1] Staring Into The Gorge: Router Exploits. http://www.renesys.com/blog/2009/08/staring-into-the-gorge.shtml.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining Specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[3] S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *CCS*, 1999.

[4] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *AIPL*, 1977.

[5] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and Enforcement of Data Structure Consistency Specifications. In *ISSTA 2006*, Portland, ME, USA, July 18–20, 2006.

[6] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: a General Approach to Inferring Errors in Systems Code. In *SOSP*, 2001.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, Feb 2001.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[9] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *NSDI*, 2005.

[10] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global Comprehension for Distributed Replay. In *NSDI*, 2007.

[11] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing Extensible IP Router Software. In *NSDI*, 2005.

[12] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE*, 2002.

[13] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[14] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[15] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[16] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[17] D. Pei, X. Zhao, D. Massey, and L. Zhang. A Study of BGP Path Vector Route Looping Behavior. In *ICDCS*, 2004.

[18] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using Queries for Distributed Monitoring and Forensics. *SIGOPS Oper. Syst. Rev.*, 40(4):389–402, 2006.

[19] M. Vaziri and G. Holzmann. Automatic Detection of Invariants in SPIN. In *Fourth Int'l SPIN Workshop*, 1998.

[20] T. N. Win, M. D. Ernst, S. J. Garland, D. Kirli;, and N. A. Lynch. Using Simulated Execution in Verifying Distributed Algorithms. *Int. J. Softw. Tools Technol. Transf.*, 6(1):67–76, 2004.

[21] M. Yabandeh, A. Anand, M. Canini, and D. Kostić. Almost-Invariants: From Bugs in Distributed Systems to Invariants. Technical report, EPFL, http://infoscience.epfl.ch/record/141383, 2009.

[22] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.