AVATARs for Pennies: Cheap N-version Programming for Replication

Atul Singh Nishant Sinha Nitin Agrawal
{atuls,nishants,nitin}@nec-labs.com
NEC Laboratories, Princeton

Software systems fail; distributed systems fail in worse ways [20]. The causes of failures can be varied, including device and hardware failures, software bugs, memory errors, and complexity of protocols. Some lead to *fail-stop* errors that bring the system (or a single node) down, while others lead to more insidious *fail-stutter* [2] or *fail-silent* errors that cause unexpected behavior. Many tools exist to find bugs in distributed systems [15]; however, bugs still remain and inevitably manifest as faults.

Replicated state machine (RSM) paradigm [22] is a practical technique to tolerate faults. The service logic is modeled as a deterministic state machine and instantiated on multiple replica machines. A consensus protocol is used to assimilate the correct result assuming no more than a threshold of faulty replicas[1]. RSM approach assumes that replica failures are *uncorrelated*—otherwise the fault threshold may be violated, affecting the safety and liveness properties of the service.

A classic approach to achieve fault-independence is to use N-version programming (NVP) [3]. Based on the same system specification, several development teams work independently to design and implement N versions. The failure diversity in such systems stems from two sources: *human* (*e.g.*, different choices of algorithms) and *programmatic* (*e.g.*, choice of language, compiler, run-time). Unfortunately, NVP has serious drawbacks: it is often prohibitively expensive, and the total time to develop a software can increase significantly, adversely affecting the time-to-market. Consequently, NVP is primarily limited to mission-critical software [4].

A middle ground for using the N-version approach is to opportunistically leverage existing diversity in software implementations for a given system specification. For example, EnvyFS [5] uses multiple open-source file system implementations (e.g., ext3, JFS, ReiserFS) as replicas while Shepherd [26] uses MySQL, Oracle, and DB2 as the replicas. A key advantage of these approaches is the relatively low cost since the base implementations are already available; unfortunately, this also means that it is not suitable for building new applications.

In this paper, we propose `AvatarFactory` as a means to achieve the necessary diversity for building reliable distributed systems. The key contribution of `AvatarFactory` is in developing an automated and cost-effective methodology to introduce the diversity; it does so in multiple ways: it exploits the diversity in off-the-shelf compilers and their corresponding *run-times*, leverages insights from existing software recovery techniques to tolerate deterministic bugs, and employs recently developed high-level domain specific languages to reduce the chances of software errors during the implementation stage. Our preliminary evaluation shows that the versions produced by `AvatarFactory` are diverse enough to tolerate compiler introduced errors.

Although `AvatarFactory` is a cost-effective approach to diversity, it is less powerful than traditional approaches to N-version programming because the initial design is developed by a single team (as opposed to N teams). However, `AvatarFactory` attempts to regain the diversity lost due to the absence of multiple human teams through a series of transformations. Note that `AvatarFactory` is not a panacea since it is impossible to automatically introduce enough diversity to mask all possible execution errors, such as resource leaks.

1 Background

Software Fault Recovery: Checkpoint and restart [11] is a widely used approach to recover from software faults but is not effective for deterministic errors. Rx [19] uses checkpoint and restart techniques in addition to changing the execution environment (via memory layout, delaying freeing pointers, zero-filling buffers, etc.) during re-execution to tolerate deterministic bugs. Microreboot [8] reduces the unavailability penalty incurred by whole-system restart approaches [12] by rebooting only the failed component, but requires the software to be designed in a loosely-coupled fashion. We explore techniques similar to Rx to improve software diversity.

Model-based design (MBD): In this approach [21], a model of the system is designed which is then tested using high-level specifications and then automatically translated to a deployed system. MBD is the basis of modern avionics and automotive systems design (e.g., based on MATLAB Simulink models) and also the electronic design automation industry. We apply the MBD approach to achieve software diversity cheaply.

Domain Specific Languages (DSL): Recently, declarative data-driven programming models, *e.g.*, based on

[1] Depending on the consensus protocol, the RSM approach can be used to tolerate crashes as well as more complex Byzantine faults.

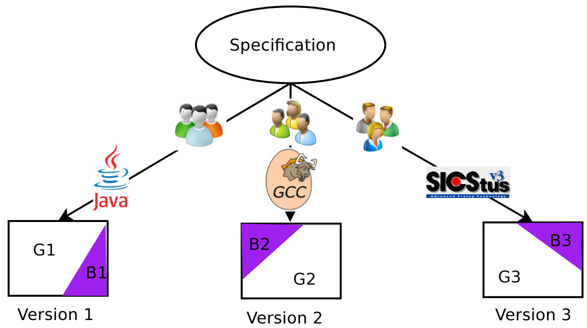

Figure 1: Understanding diversity.

DataLog [16, 27], have been proposed for implementing distributed systems. Erlang [1], a DSL for highly concurrent distributed applications communicating via asynchronous message passing, has been widely used at Ericsson and recently at Facebook. Go [24], a DSL recently developed by Google, aims to simplify programming large-scale concurrent systems. TLA+ [14] combines temporal logic with logic of actions for specifying concurrent systems.

2 `AvatarFactory` Architecture

In Figure 1, three executable software versions are generated from a single system specification. The behavior of each version consists of both *good* (G_i) and *buggy* behaviors (B_i). Bugs occur either due to erroneous human interpretation of the specification (H_i) or due to the compiler during generation of the executable (C_i), *i.e.*, $B_i = H_i \cup C_i$. For fault independence, we require that $B_i \cap B_j = \phi$ for $i \neq j$. With N different human teams and each using different development frameworks, traditional NVP approaches minimize the correlation between B_i's. Our goal is to achieve software diversity as close as possible to NVP without the due cost.

2.1 Design

The `AvatarFactory` architecture is presented in Figure 2. The system is specified by a single human team which serves as the input to `AvatarFactory`. The first step is to use testing or formal verification frameworks to reduce the likelihood of errors in the input. Once verified, the input is translated to multiple (and different) programming languages through source-to-source translation. Finally, the translated programs are compiled (or interpreted) using their respective compilers (or interpreters).

Choice of Input Language There are several choices for the input language such as Prolog, DataLog, Erlang, or TLA+. Choosing such high-level or DSLs provide multiple advantages. First, since the programs written in a DSL are succinct and close to pseudo-code, the chances of errors are reduced. Moreover, the lack of low-level details in a DSL (*e.g.*, indirect pointer accesses in C) makes

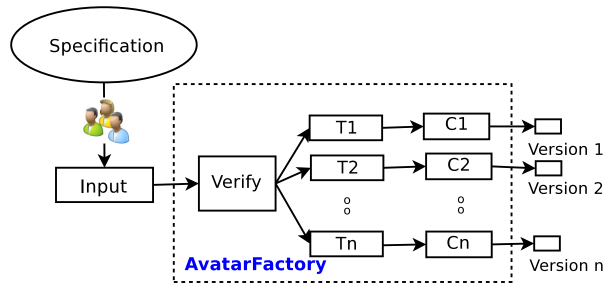

Figure 2: The `AvatarFactory` architecture. T_i represents i-th source-to-source translator and C_i represents i-th compiler/run-time to produce i-th version.

the design more amenable to rigorous formal verification techniques [10]. Therefore, using a high-level DSL as input improves the effectiveness of our framework. Moreover, recent work has shown that declarative languages (*e.g.*, DataLog [16], Prolog [17]) are quite effective at modeling distributed protocols.

Testing and Verification of Specification A large body of work exists on testing and verification of computer programs, both for traditional imperative languages to build distributed systems [10, 13] and emerging DSLs such as DAHL [17] and NDLog [27]; `AvatarFactory` can re-use these techniques directly.

Source-to-Source Language Translators Once verification is complete, a set of source-to-source translators are used to generate multiple versions of the input in different programming languages. For example, an input program in Prolog can be translated to a C and a Java source program. These translators play a key role in `AvatarFactory`; in fact, the usefulness of `AvatarFactory` crucially depends on the correctness, availability, and diversity of such translators. Fortunately, a variety of source-to-source translators exist for different source programming languages [6, 9, 25] and checkers exist to test semantic equivalence of the translated and source programs [7].

Compilers and Run-times The generated multiple source versions from the previous phase are then compiled to run directly on the machine or to be interpreted by the language run-time (*e.g.*, JVM). The compilers use a wide range of optimizations to improve the performance, such as speculative branch prediction and producing branch-free code. These optimizations enable diverse behaviors, even for the same programming language (*e.g.*, C program compiled using `-O0` and `-O3`), and thus can be useful to further improve diversity.

2.2 Regaining Lost Diversity

Note that `AvatarFactory` starts with a single high-level specification to ensure full automation and to obtain multiple versions cheaply. As a result, it may lose the diversity that multiple human teams provide in traditional NVP. In other words, since H_i's across the versions are identical for AvatarFactory, the functional errors in the input may appear as deterministic bugs in all the different versions. Although it is impossible to completely eliminate these bugs, AvatarFactory tries to reduce the incidence of correlated bugs in multiple ways.

First, by using a high-level DSL, many low-level bugs, e.g., related to memory and resource management, are avoided. Second, by rigorous verification and testing, many of these functional errors are detected and eliminated. Finally, our translation phase, coupled with the different compilers, provides an effective mechanism to make the execution environment diverse.

Different translators typically layout different data structures in the output source, leading to different behaviors. Moreover, they may have different memory allocation/deallocation schemes which result in different execution behaviors. For example, a translator to C++ may explicitly call a `class destructor`, while another translator to Java may call the `finalize` class method. The destructor frees the memory immediately while the `finalize` method is called during the next garbage collection cycle; this may lead to the resource being available longer in the Java implementation, thus avoiding a potential crash that the C++ program may succumb to. Our insights to introduce diversity in the translator phase are similar to those employed in Rx to tolerate deterministic bugs in a *single* source implementation.

3 Preliminary Evaluation

We have built an initial prototype of AvatarFactory wherein the input specification is provided in Prolog. To produce the three versions, the input is fed to the SICSTUS [23] compiler and runtime, the gprolog [9] compiler, and PrologCafe [6] system[2]. We have not yet implemented the verification phase.

We studied the bug logs of gprolog compiler and discuss only one bug [18] here due to space limits.

```
bug :-  ( catch(X is a, _, fail);
        dummy(_,s(_)), X=0 ),
        dummy(X).
dummy(_).
dummy(_,_).
test_bug :- (bug -> write('OK!\n');
            write('BUGGY!!!\n')).
:- initialization(test_bug).
```

The expected output of this code is 'OK!', however, the reporting user obtained 'BUGGY!'. This bug is triggered due to an erroneous optimization in gcc, which removes an assignment done to the ebx register, assuming that it is a local variable. The suggested fix was to reconfigure gprolog with `--disable-regs` option.

The above snippet of code was provided as input to AvatarFactory. To reproduce the bug in gprolog, we used an earlier version 1.3.0 (the bug is fixed in current version 1.3.1). The SICSTUS and PrologCafe versions did not produce the erroneous output, providing initial evidence that AvatarFactory generates diversity in the replicas to tolerate compiler introduced bugs. Going forward, we plan to complete the implementation and experimentally evaluate AvatarFactory.

[2] PrologCafe translates the input Prolog to Java which is then compiled using OpenJDK 1.6 Java compiler.

References

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, Raleigh, NC, 2007.

[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *HotOS VIII*, pages 33–38, Schloss Elmau, Germany, May 2001.

[3] A. A. Avižienis and L. Chen. On the Implementation of N-Version Programming for Software Fault Tolerance During Execution. In *COMPSAC'77*, Chicago, USA, 1977.

[4] A. A. Avižienis, M. R. Lyu, and W. Schütz. In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software. In *FTCS '88*, Tokyo, Japan, June 1988.

[5] L. N. Bairavasundaram, S. Sundararaman, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Tolerating File-System Mistakes with EnvyFS. In *USENIX '09*, San Diego, CA, June 2009.

[6] M. BANBARA and N. TAMURA. http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/.

[7] C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. Tvoc: A translation validator for optimizing compilers. In *CAV*, 2005.

[8] G. Candea, G. C, S. Kawamoto, Y. Fujiki, A. Fox, and G.Friedman. Microreboot - a technique for cheap recovery. In *OSDI*, 2004.

[9] D. Diaz. http://www.gprolog.org/.

[10] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[11] E. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. A survey of rollback-recovery protocols in message passing systems. *ACM Computer Surveys*, 2002.

[12] J. Gray. Why do computers stop and want can be done about it? In *SRDS*, 1986.

[13] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI '07*, 2007.

[14] L. Lamport. https://research.microsoft.com/en-us/um/people/lamport/tla/book.html.

[15] X. Liu, W. Lin, A. Pan, and Z. Zhang. Wids checker: Combating bugs in distributed systems. In *NSDI'07*, 2007.

[16] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP '05*, 2005.

[17] N. Lopes, J. Perez, A. Rybalchenko, and A. Singh. Applying prolog for development of distributed systems. In *ICLP*, 2010.

[18] C. optimization bug. http://lists.gnu.org/archive/html/bug-prolog/2008-09/msg00009.html.

[19] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05*.

[20] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI '06*.

[21] T. Schattkowsky and W. Mller. Model-based design of embedded systems, 2004.

[22] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[23] SICStus Prolog. http://www.sics.se/isl/sicstuswww/site/index.html.

[24] The Go Programming Language. http://golang.org.

[25] T. A. J. to C translator. http://www.cs.arizona.edu/projects/sumatra/toba/doc/.

[26] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP '07*, 2007.

[27] A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu. Formally verifiable networking. In *HotNets*, 2009.