Determining Software Models That Are Less Incorrect

Leona F. Fass

Abstract

We describe our theoretical behavioral modeling research in specific domains where finite-state models really can be found. There, with precise model-based design and perfect model checking, we obtain correct results by constructive inference, adequate testing and verification. In actual software development we find these results can only approximately apply. Still we show they can assist in software design that is "less incorrect", consistent with the results of theoreticians and practitioners working in actual model-based design and model checking.

Introduction: Our Approach to Modeling

Model-based software design is a formal approach to developing better specifications and thus, better software. Model checkers, on the other hand, are formal tools that can assist in determining whether (and where) a specification, or specified software, may be (in)correct. Such formal methods are beneficial to software designers and developers, providing mathematical foundations for systematic discovery of software defects. Since we believe that complex software designs always will have errors and flaws, we concur with Cherniavsky's "Popperian" view (Cherniavsky 1987) that discovery of defects is a positive event. For, once detected inevitable defects can be corrected, leading to development of better software…with behavior that is "less incorrect."

Techniques and problems related to designing and developing software have fallen within our investigative scope as we have sought generalized methods for correctly representing or modeling knowledge. Our interest developed as we devised a theory of learning, which might require acquisition of an infinite body of knowledge by finite means. For example, a language may consist of a (possibly) infinite set of sentences, all of which might be represented by a finite syntactic model. Such a generative model might be a finite grammar that could produce all, and only, the language's sentences. Such a recognitive model might be a finite-state device accepting any sentence that is in the language, and nothing that is *not.* If precisely the language is the correct behavior of the grammar or recognizer, then we consider acquisition of either model to be acquisition of the entire language, by finite means. Once the model is acquired the (possibly) infinite set of the language's sentences is acquired, in the sense that it is completely characterized.

We might also consider a formula or finite set of computational rules to be the finite means of acquiring an infinite body of mathematical knowledge. E.g., by learning a correct, finite set of rules for adding integers, an infinite table of sums is theoretically acquired. For, given any integers, it would be known how to compute their sum correctly, even if that specific addition problem had never been observed before.

As another example we might consider the design of an actual, physical machine to be the representation of a never-ending behavioral stream. A finite vending machine, correctly designed, could "perpetually" produce output beverages for input funds. (Today's automated machines signal for service calls when they recognize their own maintenance needs!) An elevator in a three-story building could be correctly designed as a finite-state device, transitioning from state-to-state, as it travels trip-after-trip, day-after-day, from floor-to-floor.

Similarly, a finite correct program or software system might represent an infinite set of operations or computations, each of which achieves a specified program- or system-goal. This could be the case for a simple program designed to take *any* two input integers and compute their sum; or industrial applications with domains considered by (Iscoe *et al* 1992); or the complex business systems with nonterminating e-commerce processors described in (Wang *et al* 2000). If a behavioral specification could be precisely developed, and software constructed to be correct (i.e., fulfilling that specification) then a *given* program or system might be interpreted as a finite model of an infinite body of knowledge. For, that software *itself* would represent, finitely, every possible example of how it should behave.

Here we describe our theoretician's approach to correctly determining behavioral models, beginning with some domain-specific results noted in (Fass 1989, Keller 1992, Fass 2000) and proceeding to our attempts to adapt our work to the general area of software design. We first showed that within a particular problem domain a finite model could be inductively constructed, adequately tested, and finite-state verified, in the sense we describe. We found these results should *theoretically* assist in developing correct software, for *if* a complete behavioral specification exists, and *if* finite correct fulfilling software exists, then our techniques should determine it, effectively. But as we discuss, we inevitably discovered that in *reality* our results can only approximately apply. Hence we describe how our model-based design, and our version of model checking for correctness, at least can establish that some software is less incorrect. We conclude that our findings compare favorably with those

of theoretical researchers and practitioners who have sought correct software through model-based development and model checking.

Our Original Modeling Results

We initially had great success designing finite models of knowledge within a constrained formal linguistic domain, determining syntactic models from behavioral examples. There we showed that an entire linguistic behavior could be precisely specified, and a finite language model effectively constructed, from a sample of how it *should* behave. We then showed that a potential model could be conclusively tested to determine, from a sample of how it *should* and should *not* behave, whether (or not) it is correct.

To obtain these results we analyzed some of the structures within the infinite linguistic domain, to determine what distinguishes those elements that are structures in a specific (CF) language from those which are not. We then showed that for *any* element of the structural domain, it could be decided whether that element was in the "correct" (i.e., within the language structures) or "incorrect" (i.e., within the complementary domain structures) behavioral set. Next we showed that, based on behavioral membership, the entire linguistic domain could be partitioned into a *finite* set of congruence classes. Each class consisted of *all* domain elements that, relative to the specified linguistic behavior, "always behaved in the same way" (They could indistinguishably appear within the same linguistic structures). *Every* domain element was a member of exactly one such behavioral congruence class. By determining the classes we could precisely and *completely* specify the infinite linguistic behavior by finite means.

We were able to establish that for any such specified behavior a finite (grammatical or recognitive) syntactic model *exists* and may be effectively constructed, with components corresponding to the behavioral classes just described. We then showed that a specific minimal finite model could be inductively inferred, and constructed, from an appropriate, finite, correct behavioral sample. (Since a finite set of congruence classes completely specifies the linguistic behavior, a finite set of *representatives* of each of the classes proves to be sufficient information for construction of a minimal model). The syntactic model provides another way to specify the infinite linguistic behavior by finite means.

Motivated by (Cherniavsky 1987) we next showed that a correct behavioral model might also be determined through effective tests. For, within our constrained linguistic domain, we found that not only could we finitely represent all correct behavior, we could also finitely represent all behavior that was *not* correct. We had just defined that finite correct behavioral sample from which a minimal model could be found. We now determined that this sample, and its finite relative complement (within the constrained domain) provided sufficient positive (correct) and negative (complementary) examples to define an adequate, conclusive set of tests.

If a *potential* minimal model failed to produce all of the positive behavior in the specified examples, or produced any of the complementary, negative behavior specified, incorrectness would be detected, and the potential model could be revised. But if no specified experiment detected incorrectness, our process provably *guaranteed* the potential model to be correct. Finitely representing all the ways the potential model might fail, and finding none, by default the model would be finite-state verified (Clarke 2000).

Thus in our initial research involving the constrained linguistic domain (as detailed in Fass 1989, Fass 2000), we established that a minimal behavioral model *exists*, and that there are many ways to determine it. *If* the model can be inductively constructed from a finite positive (correct) behavioral sample, *then* a potential such model may be adequately tested for incorrectness and, perhaps, by default verified. Furthermore, the finite positive behavioral sample sufficient for inference determines the finite (positive and negative) behavioral sample for conclusive, effective tests.

Once we obtained these "perfectly correct" domain-specific modeling results we attempted to generalize them to other problem areas. Next we describe our discoveries, as we sought to similarly infer, test or verify "perfectly correct" software, from specified behavioral-domain examples.

Adapting Our Research to Software Design

When we sought to adapt our techniques for inference/testing/verification to the application area of software design, we realized why our original (linguistic domain) results had been so satisfying. The constraints we had imposed to convey the (linguistic) behavior actually had *forced* our modeling results. With these constraints the design problem became *completely specifiable* (every behavioral element corresponded to some model component) and the behavior *finitely realizable* (hence the finite characterizing model exists!) with membership decidable (hence correct behavior is distinguishable from its complement). Our domain constraints enabled us to characterize all possible positive behavior, and all possible negative behavior by finite means.

Within the present context we classify our "perfect" design approach as *model-based.* For, we analyze the behavioral domain to determine its finite (perhaps forced) underlying structure. We then exploit the structure to determine a finite characterization. Our testing and/or verification approach is "perfect" *model checking*, in the sense that *all* possible behavior is represented in a finite

(positive and negative) sample so that the comparative checking process concludes, effectively. In the constrained linguistic domain a precisely characterizing behavioral model exists, and either approach enabled us to find it. Such results are domain-specific only in the sense that they are applicable in any behavioral domain that can be completely specified by a finite device (Fass 2000).

Now, how do these theoretical results adapt in the practical application area of software development? Designing a program or complex software system that fulfills a specification is "just" a problem of constructing or determining a correct behavioral model. Any *given* software producing a specified behavior is obviously a finite entity or "device". It would be nice to infer such software from a finite sample of the correct behavior, as specified. If we could, we should then also be able to adequately test potential software from a finite characterizing sample of correct behavior and its complement (also defined by the specification). Hence software could be "verified".

But the constraints applied in our theoretical work do not exist in "real life" software development and design. Furthermore many decisions we would like to make in assessing software correctness just cannot be made. It is well know that the behavioral equivalence of a program and its specification is generally undecidable. A specification is never complete, if it exists at all. And in general there is no adequate test set that can conclusively determine if software is incorrect. When "real life" testing detects no errors, this does not mean that the software is validated or verified. More likely, this means that there is something wrong with the tests (Arthur 1999, Cherniavsky 1987, Gargantini and Heitmeyer 1999, Wang *et al* 2000, Weyuker 1983). Faced with such obstacles, software developers do the best that they can.

Our constrained inference/testing/verification results can assist current software developers even without guaranteeing correctness, for we can rigorously show some software components to be "approximately correct". That is, if a subcomponent of a software system really *is* representable as a finite-state device, we can show there is a behavioral data sample from which that subcomponent can be inferred. If correct and incorrect behavior can be distinguished, we can define a data sample with which a potentially correct subcomponent can be adequately tested. This was essentially shown by (Weyuker 1983) for programs computing arithmetic problems, only for mod 4. Another example might be the lexical analyzer component of a compiler, typically represented as a finite-state machine. We are able to infer and test such, given enough computation space and time.

Now, such successful inference would only produce a software subcomponent that is correct, thus *approximating* correctness in the greater software system. Testing a subcomponent, relative to the greater system would only be "approximate testing". Detecting and correcting subcomponent defects might verify the subcomponent. But in relation to the greater software system, it would, at best, establish the software to be "less incorrect".

We believe that "approximate correctness" based on formal mathematical foundations (as we have described) is an improvement on unfounded, ad hoc, software development results. If such approximating correctness can determine that software is "less incorrect", we consider this a better result than ad-hoc assessment of software design, not provably correct at all.

Conclusions: Related Research

We are gratified to see our beliefs confirmed in recent *formal* approaches to software development, in research of practitioners as well as theoreticians. The well-intentioned quest for perfectly correct software has been gradually replaced by the goal of *improving* design, or at least detecting and correcting more errors. We find this in the areas of specification (Gunter *et al* 2000, Wing 1992) and model-based design (Aida *et al* 1999, Iscoe *et al* 1992, Keller 1992); model checking (Clarke *et al 1989*, Emerson and Lei 1986, Gargantini and Heitmeyer 1999, Henzinger *et al* 1992, Wang *et al* 2000) and validation/verification (Arthur *et al* 1999, Clarke 2000, Moore 2000, Vardi 1987).

Researchers eliciting user requirements to develop specifications find they are delimited by properties of varying system environments, and subjected to human (user) commission/omission errors. Testers and simulators find they can never categorize all correct behavior, let alone all the ways a system can go wrong. Verifiers and theorem-provers, even for finite-state software approximations, prove to involve such time-consuming human intervention, they cannot be put to practical use. *Automated* model checkers are useful for verifying certain system properties such as *safety* and *liveness* (Gunter *et al* 2000, Henzinger *et al* 1992, Wang *et al 2000*), and may force finite-stateness to obtain mathematically provable results. But, due to formula size in symbolic checkers, and "the state explosion problem" in explicit-state checkers, even this approximating technique is constrained by computational space and time.

Thus researchers seeking software coverage to determine *correctness* find model checking is best reserved for system subcomponents that are mission-critical (Clarke 2000, Gargantini and Heitmeyer 1999, Wang *et al* 2000). They are satisfied, instead, when a model checker detects specification errors and traces software faults. As in our own modeling research, they (Gargantini and Heitmeyer 1999, Moore 2000, Wang *et al* 2000) find that a process to determine correctness *also* defines error-detection tests.

When J. Strother Moore (Moore 2000) was asked how his recent automated verification process for hardware components differed from his earlier verification of software, he noted there was *no* difference. In either case it was not the entity (hardware, software) being verified; what was verified was its mathematical model! So, even when mathematical foundations can conclusively establish correctness of a model, this is *still* an approximating result. Thus, we believe it is a realistic goal to develop software with the foundation of a sound mathematical model that guarantees it to be *less incorrect.* The adaptation of our original modeling results, as described above, should help to achieve that goal.

References

Aida, T. *et al*. 1999. "Model-Based Specification and Generation of Programs", *AAAI-99 Workshop on Intelligent Software Engineering*, Orlando, FL July 1999, 1-6.

Arthur, J.D. *et al*. 1999. "Evaluating the Effectiveness of Independent Verification and Validation", *IEEE Computer*, Vol 32, No 10, October 1999, 79-83.

Cherniavsky, J.C. 1987. "Computer Systems as Scientific Theories: a Popperian Approach to Testing", *Proc. Of the 5th Pacific Northwest Software Quality Conf.*, Portland OR, October 1987, 297-308.

Clarke, E. *et al*. 1989. "Compositional Model Checking", *Fourth Annual Symposium on Logic in Computer Science (LICS)*, IEEE, June 1989, 353-362.

Clarke, L., 2000. "Finite State Verification: An Emerging Technology", abstract of Invited Presentation, *SIGSOFT 2000 Intl Symposium on Software Testing and Analysis*, Portland OR, August 2000, *ACM SEN* Vol 25, No 5, 146.

Emerson, E. A. and C-L Lei, 1986. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus", *First LICS Symposium*, IEEE, June 1986, 267-278.

Fass, L.F., 1989. "A Common Basis for Inductive Inference and Testing" *Proc. of the 7th Pacific Northwest Software Quality Conf.*, Portland OR, September 1989, 183-200.

Fass, L.F. 2000. "Establishing Software 'Correctness' by Logical and Algebraic Means", *Proc. of the 5th Joint Conf on Information Sciences*, Atlantic City, NJ, February 2000, Vol I, 639-642.

Gargantini, A and C. Heitmeyer, 1999. "Using Model Checking to Generate Tests from Requirements Specifications", *7th European Software Engineering Conf/ ACM SEN* Vol 24, No 6, November 1999, 146-162.

Gunter, C. *et al*, 2000. "A Reference Model for Requirements and Specifications", *IEEE Software*, Vol 17, No 8, May/June 2000, 37-43.

Henzinger, *et al*, 1992. "Symbolic Model Checking for Real-time Systems" *Seventh Annual LICS*, IEEE, June 1992, 39-406.

Iscoe, *et. al.*, 1992. "Model-Based Software Design", appears in (Keller, 1992), 72-77.

Keller, R (Ed.) 1992. *Notes of the AAAI Workshop on Automating Software Design: Domain Specific Software Design*, San Jose CA, July 1992, NASA Ames Research Center AI Research Branch TR FIA-92-18.

Moore, J.S, 2000. "Machines Reasoning about Machines", Invited Presentation, *AAAI 2000*, Austin TX, August 2000 and private communication September 2000.

Vardi, M. 1987. "Verification of Concurrent Programs: The Automata Theoretic Framework", *Second LICS*, IEEE, June 1987, 167-176.

Wang, W. *et al*, 2000. "E-Process Design and Assurance Using Model Checking", *IEEE Computer*, Vol 33, No 10, October 2000, 48-52.

Weyuker, E.J., 1983. "Assessing Test Data Adequacy through Program Inference", *ACM Trans. On Programming Languages and Systems*, Vol 5, (1983), 641-655.

Wing, J.M. 1992. "Specifications in Software Development", *Seventh Annual LICS*, IEEE, June 1992, 112.

Leona F. Fass received a B.S. in Mathematics and Science Education from Cornell University and an M.S.E. and Ph.D. in Computer and Information Science from the University of Pennsylvania. Prior to obtaining her Ph.D. she held research, administrative and/or teaching positions at Penn and Temple University. Since then she has been on the faculties of the University of California, Georgetown University and the Naval Postgraduate School. Her research primarily has focused on language structure and processing; knowledge acquisition; and the general interactions of logic, language and computation. She has had particular interest in inductive inference processes, and applications/adaptations of inference results to the practical domain. She may be reached at

Mailing address: P.O. Box 2914
Carmel CA 93921
lff4@cornell.edu