A Theorem Proving Approach for Verification of Reactive Concurrent Programs

Sandip Ray
University of Texas at Austin
sandip@cs.utexas.edu

Rob Sumners
Advanced Micro Devices Inc.
robert.sumners@amd.com

***Abstract*— We present a framework for the specification and verification of reactive concurrent programs using general-purpose mechanical theorem proving. We define specifications for concurrent programs by formalizing a notion of refinements analogous to stuttering trace containment. The formalization supports the definition of intuitive specifications of the intended behavior of a program. We present a collection of proof rules that can be effectively orchestrated by a theorem prover to reason about complex programs using refinements. The framework is integrated with the ACL2 theorem prover and we demonstrate its use in the verification of several concurrent programs in ACL2.**

I. Overview

Reactive concurrent programs consist of interacting processes which perform ongoing, non-terminating computations while receiving stimulus from an external environment. The complexity induced by these interactions makes concurrent programs particularly error-prone, with bugs difficult to detect and diagnose. With pervasive deployment of multicore and multiprocessor systems, there is critical need for robust methodologies for verification of reactive concurrent programs.

This paper presents a verification framework for concurrent programs in a general-purpose theorem prover. We model concurrent program implementations as (possibly unbounded-state) labeled transition systems, and our verification methodology entails proving that the implementation is a refinement of a simpler system that serves as the specification. We develop (1) a formal *notion of correspondence* for relating implementation and specification, (2) a collection of *formalized proof rules* that reduce the verification to the definition and proof of an invariant, and (3) an *integrated predicate abstraction tool* to automate the latter proof. Our framework is mechanized with the ACL2 theorem prover.

The notion of correspondence we use is *stuttering trace containment*: "For each (infinite) execution of the implementation there is an (infinite) execution of the specification with the same observable behavior up to finite stuttering." Stuttering trace containment and related notions of stuttering simulation and bisimulation have been studied extensively in the context of semantics for reactive systems [1], [2]. However, there has been little work on formalizing the notion in a theorem prover for verifying program implementations. Instead such notions have been argued informally (*e.g.*, to metatheoretically justify proof rules built into the reasoning process [3], [4], [5].) Unfortunately, that approach does not afford sound extension of the repertoire of proof rules. With our framework, one can verify a new proof rule and use it subsequently in the verification of concrete programs. We found extensibility critical for handling diverse programs each with its own unique idiosyncrasies.

Developing the framework in ACL2 is non-trivial, in part because of ACL2's limited expressivity. The logic is first-order, and does not permit infinitary objects. The logic does allow partially defined, constrained, and uninterpreted functions, which can be used to simulate a limited amount of higher-order reasoning sufficient for our purpose. In our work, logical limitations manifest in the formalization of trace containment (cf. Section II). However, once this formalization is completed, verification of individual programs reduces to a first-order problem for which ACL2 is suitable. In addition, ACL2 provides several features (*e.g.*, an ANSI-standard programming language as the formal language for defining systems, support for efficient simulation, and the ability to handle large formulas) which are invaluable for scalability.

Preliminary scripts supporting the work described here are distributed with ACL2 in the directory `books/concurrent-programs`.

II. Stuttering Trace Containment in ACL2

A reactive system M is described by three functions, namely $M.\mathit{init}()$, $M.\mathit{next}(s, i)$, and $M.\mathit{label}(s)$, which correspond to the initial state, state transition function (that takes a current state s and an environmental stimulus i), and state labeling function respectively. Given a unary function env such that $\mathit{env}(k)$ represents the stimulus at time k, the function $M.\mathit{exec}[\mathit{env}](n)$ below returns the state of M after n transitions.

Definition Schema.

$$M.\mathit{exec}[\mathit{env}](n) \triangleq \begin{cases} M.\mathit{init}() & \mathit{if}\ (n = 0) \\ M.\mathit{next}(M.\mathit{exec}[\mathit{env}](n-1), \mathit{env}(n-1)) & \text{otherwise} \end{cases}$$

The use of the unconventional names above illustrates one of the "tricks" involved in formalizing reactive systems in the logic of ACL2. In ACL2 logic, an infinite sequence of inputs must be modeled as a *function* over a natural-valued time; however, since functions in the logic cannot take arbitrary functions as arguments, the notion of an infinite execution of a system must be written as a schema rather than as a single, closed-form definition. A consequence is that for each function *env*, the function $M.\mathit{exec}[env]$ represents an execution of system M.

Details of the translation of a concurrent program definition into this formulation are beyond the scope of this paper. But the basic idea is that for a given concurrent program, the definitions of $M.\mathit{init}$ and $M.\mathit{next}$ would correspond to the initial state and state transformations derived from the execution of the concurrent program within the context of the formal semantics of the programming language. The definition of $M.\mathit{label}$ is determined by the components of the program relevant to the notion or specification of correct execution of the concurrent program.

Trace containment relates *each* execution of an implementation I with *some* execution of the specification S. We formalize this notion as follows. Let *uenv* be an uninterpreted function; thus, $I.\mathit{exec}[\mathit{uenv}]$ represents an arbitrary execution of I. Ignoring stuttering for the moment, proving trace containment reduces to the obligation of defining a (concrete) function *cenv* such that the following condition is satisfied.

Trace Containment Obligation.
$I.\mathit{label}(I.\mathit{exec}[\mathit{uenv}](n)) = S.\mathit{label}(S.\mathit{exec}[\mathit{cenv}](n))$

We augment the above notions with provision for stuttering by restricting the stimulus functions. Informally, we view the environment stimulus *env* as a pair of functions $\langle \mathit{stim}, \mathit{ctr}\rangle$, where the second component is simply a counter controlling the number of stuttering steps and the first component is the actual stimulus. Formally, given a system M, the notion of a stuttering trace of M for the environmental stimulus is given by $M.\mathit{trace}[stim, ctr]$ below.

Definition Schema.
$$M.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n) \triangleq \begin{cases} M.\mathit{init}() & \text{if } n = 0 \\ M.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n-1) & \text{if } \mathit{ctr}(n) \prec \mathit{ctr}(n-1) \\ M.\mathit{next}(M.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n-1), \\ \quad \mathit{stim}(n-1)) & \text{otherwise} \end{cases}$$

Here $\prec$ is a well-founded relation. $M.\mathit{trace}$ is analogous to $M.\mathit{exec}$, except that when $\mathit{ctr}(n-1) \prec \mathit{ctr}(n)$, $M.\mathit{trace}$ stutters, ignoring the stimulus. Well-foundedness of $\prec$ guarantees that stuttering is finite.

The notion of stuttering trace containment is then given by the following proof obligation; as with trace containment, *ust* is an uninterpreted function and the obligation is to come up with *cst*. We will use $(S \rhd I)$ to mean that S is a refinement of I under this obligation.

Stuttering Trace Containment Obligation.
$I.\mathit{label}(I.\mathit{trace}[\mathit{ust,ctr}](n)) = S.\mathit{label}(S.\mathit{trace}[\mathit{cst,ctr}](n))$

The definitions above are difficult to use directly in the logic of ACL2 for reasoning about concurrent programs. Fortunately, we can develop proof rules to alleviate the problem. Among the rules available in our framework are the following:

- **Stepwise Refinement Rule** reduces the obligation $(S \rhd I)$ to the definition of an intermediate model M and showing $(S \rhd M)$ and $(M \rhd I)$.
- **Single-step Refinement Rule** reduces the proof of $(S \rhd I)$ to a collection of proof obligations that do not require reasoning about more than one transition of any system (see below).
- **Local Reduction Rule** collapses local transitions (transitions involving only local variables of a process in a system) to a single atomic step.
- **Oblivious Rule** permits augmenting a system with auxiliary history and prophecy variables.
- **Pipeline Rule** replaces overlapped concurrent executions of successive transitions into a sequence of transitions executed in program order.

In practice, the main "workhorse" for verification of concurrent programs is single-step refinement. The rule is inspired by (and derived from) corresponding rules for well-founded bisimulation [5] and is a critical proof rule in our framework. Given two systems $\mathcal{S}$ and $\mathcal{I}$, we will say that $\mathcal{I}$ is a *single-step refinement* of $\mathcal{S}$, written $(\mathcal{S} \trianglerighteq \mathcal{I})$ if and only if there exist functions *inv*, *skip*, *rep*, *rank*, *pick*, and *good* such that the following formulas are theorems.

ST1: $\mathit{good}(s) \Rightarrow \mathcal{I}.\mathit{label}(s) = \mathcal{S}.\mathit{label}(\mathit{rep}(s))$
ST2: $\mathit{good}(s) \wedge \mathit{skip}(s,i) \Rightarrow \mathit{rep}(\mathcal{I}.\mathit{next}(s,i)) = \mathit{rep}(s)$
ST3: $\mathit{good}(s) \wedge \neg\mathit{skip}(s,i) \Rightarrow \mathit{rep}(\mathcal{I}.\mathit{next}(s,i)) = \mathcal{S}.\mathit{next}(\mathit{rep}(s), \mathit{pick}(s,i))$
ST4: $\mathit{ordinal\text{-}p}_{\prec}(\mathit{rank}(s))$
ST5: $\mathit{good}(s) \wedge \mathit{skip}(s,i) \Rightarrow \mathit{rank}(\mathcal{I}.\mathit{next}(s,i)) \prec \mathit{rank}(s)$
ST6: $\mathit{inv}(\mathcal{I}.\mathit{init}())$
ST7: $\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathcal{I}.\mathit{next}(s,i))$
ST8: $\mathit{inv}(s) \Rightarrow \mathit{good}(s)$

Single-step Refinement Rule.
Derive $(\mathcal{S} \rhd \mathcal{I})$ from $(\mathcal{S} \trianglerighteq \mathcal{I})$

Informally, given a state s of $\mathcal{I}$, $\mathit{rep}(s)$ returns a corresponding state of $\mathcal{S}$ with same label. The predicate *skip* governs stuttering. If $\mathit{skip}(s,i)$ is false then **ST3** guarantees that $\mathcal{S}$ has a transition that matches the transition of $\mathcal{I}$ from state s on input i, otherwise **ST2** guarantees that $\mathcal{S}$ can stutter. **ST4**, **ST5** and well-foundedness of the ordinals guarantee that stuttering is finite. **ST6** and **ST7** specify that the predicate *inv* is an *inductive invariant* of $\mathcal{I}$. That is, *inv* holds at $\mathcal{I}.\mathit{init}()$ and if it holds at a state s then it holds after any transition from s. **ST8** stipulates that the predicate *good* is logically implied by *inv*. Thus *good* must hold for all reachable states of $\mathcal{I}$. This allows us to assume $\mathit{good}(s)$ in the hypothesis of conditions **ST1-ST5**.

The single-step refinement rule reduces essentially the verification problem to the definition and proof of an inductive invariant *inv* as follows. The proof of $(S \trianglerighteq I)$ is broken into two phases.

1) Define *good*, *rep*, *skip*, *pick*, and *rank* and prove obligations **ST1-ST5**.
2) Define *inv* and prove **ST6-ST8**.

In practice, the first phase is relatively straightforward. For instance, assume that $\mathcal{I}$ is a multiprocess system implementing a cache coherence protocol and that $\mathcal{S}$ is a system of processes which atomically access and update the main memory. Then *good* needs to posit that the caches are coherent, *rep* projects the visible components (processes and memory) of the cache system, *skip* holds for transitions which cause no access to the memory, and *rank* counts the number of transitions before a visible component is updated. The proof obligations **ST1-ST5** required for well-founded refinements can usually be discharged with little or no manual effort once the appropriate definitions are provided.

The definition of the inductive invariant *inv* required on the second phase, however, is non-trivial. In particular, since *inv* must be preserved by every transition of I, it must characterize every reachable state. To ameliorate the difficulty, we have integrated the framework with a tool based on predicate abstraction. Predicate abstractions have been used in a number of formal verification tools for automating invariant proofs. However, our approach is designed to leverage the expressiveness and flexibility of theorem proving for discovering predicates: useful predicates are "mined" by applying term rewriting on the definition of the state transition function of the implementation. Rewriting is guided by *rewrite rules* which are taken from theorems proven by the theorem prover. Automating invariant proofs, of course, is a topic of independent research interest, and previous papers [6], [7] cover details of our implementation.

III. Fairness Constraints

Stuttering trace containment requires that *every* execution of the implementation corresponds to some execution of the specification. In some cases, we may only require the correspondence for executions satisfying some fairness conditions. Fairness requirements typically arise for showing progress properties; in our framework, this manifests itself in the proof of finiteness of stuttering. Here we only provide a sketch of our approach; a previous paper [8] provides a technical description.

Our approach for integrating fairness constraints with our framework is analogous to the approach for introducing the notion of stuttering, *viz.*, constraining the stimulus functions with appropriate constraints. The most common fairness constraint we use is *unconditional*: a stimulus function *stim* is fair if and only if for every time instant m and each legal input i, there is an instant $n > m$ such that $\mathit{stim}(n) = i$. The constraint is analogous to the weak fairness in Unity [9]. For most problems in practice, we have found this simplistic formalization sufficient. Nevertheless, the framework allows reasoning about more general notions of fairness using approaches similar to the one we describe below. For instance, our previous paper [8] discusses an integration of conditional fairness where one associates a set of legal inputs with each system state and fairness ensures that an input that is infinitely often legal is selected infinitely often.

Given the notion of a fairness, our framework permits attaching fairness constraints both to specification S and implementation I. We say that I is a refinement of S *under fairness assumption* (denoted $S \rhd_F I$) if for each fair trace of I there is a trace of S with the same observable behavior. We say that I is a refinement of S *with fairness requirement* (denoted $S_F \rhd I$) if for each trace of I there is a fair trace of S with the same observable behavior. The following proof rules are easy to verify.

- Derive $(\mathcal{S} \rhd_F \mathcal{I})$ from $(\mathcal{S} \rhd \mathcal{I})$.
- Derive $(\mathcal{S} \rhd_F \mathcal{I})$ from $(\mathcal{S} \rhd_F \mathcal{R})$ and $(\mathcal{R} \rhd_F \mathcal{I})$.
- Derive $(\mathcal{S}_F \rhd_F \mathcal{I})$ from $(\mathcal{S}_F \rhd_F \mathcal{R})$ and $(\mathcal{R}_F \rhd_F \mathcal{I})$.
- Derive $(\mathcal{S}_F \rhd \mathcal{I})$ from $(\mathcal{S}_F \rhd_F \mathcal{R})$ and $(\mathcal{R}_F \rhd \mathcal{I})$.

Furthermore, analogous to single-step refinement, we define a proof rule for single-step fair refinements that only requires reasoning about one transition of a program.

Note that while fairness *assumptions* are necessary for ensuring progress properties of implementations, fairness *requirements* are usually necessary for the purpose of composition. In particular, assume that we want to prove $(\mathcal{S} \rhd \mathcal{I})$ by introducing an intermediate model $\mathcal{R}$. If we need a fairness assumption in the proof of correspondence between $\mathcal{S}$ and $\mathcal{R}$, then chaining the sequence of refinements requires that we prove the correspondence between $\mathcal{R}$ and $\mathcal{I}$ with fairness requirement.

IV. Applications

We have used our framework for verification of several concurrent programs. Among the programs verified are the following.

a) **A concurrent deque implementation***:* We verify the concurrent deque implementation used in the work-stealing algorithm of the Hood thread library [10]. The program is subtle since all the methods are non-blocking; the non-blocking property is crucial to the efficiency of work-stealing. Previous work [10] discusses the subtlety of the implementation. Our framework is used to verify that the implementation is a refinement of a specification where each contending thread atomically steals the work from the deque; the verification entails orchestration of proof rules to chain together a series of refinements involving the specification, implementation, and two intermediate models.

b) **German cache coherence protocol***:* The German protocol has been widely used as a benchmark for many verification methodologies. In this protocol clients communicate with a home process through three channels to gain access to cache lines from a central memory. We prove that a model of the protocol (that contains an unbounded number of clients as well as a model of the memory and caches of participating processes in addition to the control signals) is a refinement of a specification system in which there is no home or cache and each client atomically accesses and updates the memory directly. The verification demonstrates the scalability of our predicate abstraction procedure. Given a collection of rewrite rules and a predicate *good* (that essentially states that the caches of participating processes are coherent), our procedure automatically proves the invariance of *good* in the order of seconds. It generates a predicate abstraction graph of 46 predicates, and reachability analysis explores roughly 7000 nodes and $300,000$ arcs. This is in stark contrast to other predicate abstraction approaches we are aware of (*e.g.*, UCLID [11] requires hours for the same verification).

c) **A bakery implementation***:* We verify a model of the Bakery algorithm, that is closely related to a microarchitectural implementation and optimized in several aspects. The specification is a simpler system that executes the critical region atomically. It turns out that there are executions of the implementation that do not satisfy progress because of a conflict between process scheduling and the protocol: a process p ready to enter the critical section may never be scheduled while other processes that are scheduled cannot make progress until p executes. This problem was discovered while attempting to prove finiteness of stuttering. Once the problem was identified, we could prove the implementation correct under fairness assumption.

d) **Leader Election Protocol***:* In *leader election*, the goal is for a collection of processes to communicate with each other to determine the identity of the process with the lowest index. We verified a standard but low-level synchronous implementation of a leader election protocol on a *token ring*. A process non-deterministically initiates the protocol by sending a token to its neighbors in the ring, and each process subsequently alternates between receiving and passing the token, with a finite number of local steps in between, until all processes reach a consensus on the identity of the leader. Our specification is a simple abstract system with two processes in which the leader is selected atomically in one transition after initiation.

e) **The Apprentice program***:* This multithreaded JVM bytecode program involves updates to a shared counter. Moore and Porter [12] put it forward as a benchmark against which to measure approaches to formally verifying Java programs. They also prove, using ACL2, that the program satisfies the *weak monotonicity property*, (*i.e.*, the counter never decreases), and show why the proof is non-trivial. Nevertheless, their proof does not ensure that the counter eventually increases. (Indeed, if a thread is never scheduled after spawning a child then the counter does not increase.) Our framework handles the progress property under assumption of fair thread scheduling; we prove that the Apprentice program is a refinement (under fairness assumption) of a program that atomically increments the counter.

In spite of the diversity of systems being verified, the notion of correspondence used is always the same, namely stuttering trace containment (possibly with fairness constraints). The use of a generic notion of correctness, formalized and mechanized proof rules that can be extended on demand, and integrated tools for automating expensive proof steps, makes the approach scalable for reasoning about complex implementations.

V. Discussion and Future Work

The development of our framework involved three key design decisions:

1) using a general-purpose theorem prover as the underlying reasoning engine;
2) using program correspondence instead of temporal logic for specifying program correctness; and
3) including a provision for stuttering in the notion of correspondence.

The choice of a general-purpose theorem prover is governed by the need for flexibility and control. A theorem prover facilitates clean decomposition of proofs, and sound extension and careful orchestration of strategies. Indeed, many of our proof rules were crafted on demand; when the available rules were found insufficient, they were restructured or new ones were added. Furthermore, as we discussed with predicate abstractions, decision procedures can be easily integrated to effectively automate expensive proof steps. Indeed, they become more effective in this context since the theorem prover can use user-proven lemmas to control the complexity of the abstraction on which model checking is applied.

Instead of defining the specification as an abstract program (that is related to the implementation via refinement) we could have chosen to specify program properties directly as temporal logic formulas. Our choice is based on a number of considerations. First, most general-purpose theorem provers (*e.g.*, ACL2, HOL, PVS) support *classical logic*; a semantic embedding of temporal logic in such a theorem prover is non-trivial; the problem is exacerbated in a first-order theorem prover like ACL2. Second, most of our target programs are parameterized models with an unbounded number of processes; temporal logic specification of such programs requires an expressive logic (*e.g.*, allowing quantification of process indices), and the meaning of the resulting specification can be error-prone due to possible nesting of quantifiers over time, branching, and design parameters. Third, informal human review of the completeness and correctness of temporal specification requires familiarity in formal logic in general and temporal logic specifically. Specification by program refinement allows the specification and implementation to be defined in the same operational language. This helps avoid errors that may otherwise be missed due to differences in semantics between the specification and the implementation.

The decision to allow for stuttering in the notion of correspondence stems from our desire to provide simple, intuitive specification. Admittedly, defining the theory of stuttering trace containment in ACL2 was complex. However, that was only performed once. For each application discussed in Section IV, the specification system was the obvious, intuitive abstraction capturing the design intent of the protocol. This is no coincidence. In practice, concurrent programs are often optimized elaborations of simpler protocols. These elaborations are designed to achieve execution efficiency, refine atomicity, match a given architecture, and so on. The simpler protocol then provides a succinct operational description of the intended behaviors of the elaborated implementation. Stuttering is used to reconcile the difference in the level of abstraction at which the implementation and specification are modeled, while limiting the amount of stuttering to be finite ensures that both safety and progress properties of the implementation are preserved in the specification. It is also worth noting that even when the specifications are defined in terms of temporal logic properties, it is common to define abstract models that hide details of the implementation and facilitate the verification process. In practice, a notion of correspondence with finite stuttering allows us to use refinements both as a specification and a proof mechanism.

Future Work*:* Future work involves two key research thrusts: making the framework more robust and extensible, and using it for more diverse systems. Towards the first goal, we are augmenting the framework with more reduction theorems and designing a better user interface to facilitate practical applications. One key planned augmentation is support for *real-time constraints*. Note that fairness constraints only ensure that a fair stimulus is eventually picked. Real-time systems require more stringent constraints, *e.g.*, that the stimulus is picked within a fixed upper bound of time. Towards the second goal, we are using the framework for reasoning about microarchitecture implementations, distributed garbage collection algorithms, and distributed checkpointing systems.

VI. Acknowledgments

The research of the first author is supported in part by the National Science Foundation under Grants CNS-0910913 and CCF-0916772, by the Defense Advanced Research Projects Agency under Grant N66001-10-2-4087, and by the Semiconductor Research Corporation under Grant 08-TJ-1849. The authors thank Matt Kaufmann, Robert Krug, and J Strother Moore for numerous comments and insights. Additionally, Krug and Moore read an earlier draft of the paper and made several useful suggestions.

References

[1] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1990.

[2] D. Park, "Concurrency and Automata on Infinite Sequences," in *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, ser. LNCS, vol. 104. Springer-Verlag, 1981, pp. 167–183.

[3] P. Attie, "Liveness-preserving Simulation Relations," in *Proceedings of 18th ACM Symposium on Principles of Distributed Computing (PODC 1999)*, J. Welch, Ed. ACM Press, May 1999, pp. 63–72.

[4] B. Jonsson, A. Pnueli, and C. Rump, "Proving Refinement Using Transduction," *Distributed Computing*, vol. 12, no. 2-3, pp. 129–149, 1999.

[5] P. Manolios, K. Namjoshi, and R. Sumners, "Linking Model-checking and Theorem-proving with Well-founded Bisimulations," in *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV 1999)*, ser. LNCS, N. Halbwacha and D. Peled, Eds., vol. 1633. Springer-Verlag, 1999, pp. 369–379.

[6] R. Sumners and S. Ray, "Reducing Invariant Proofs to Finite Search via Rewriting," in *5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, M. Kaufmann and J. S. Moore, Eds., Austin, TX, Nov. 2004.

[7] S. Ray and R. Sumners, "Combining Theorem Proving with Model Checking Through Predicate Abstraction," *IEEE Design & Test of Computers*, vol. 24, no. 2, pp. 132–139, 2007.

[8] R. Sumners, "Fair Environment Assumptions in ACL2," in *4th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2003)*, W. A. Hunt, Jr., M. Kaufmann, and J. S. Moore, Eds., Boulder, CO, July 2003.

[9] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Cambridge, MA: Addison-Wesley, 1990.

[10] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread Scheduling in Multiprogramming Multiprocessors," *Theory of Computing Systems*, vol. 34, pp. 115–144, 2001.

[11] S. K. Lahiri and R. E. Bryant, "Constructing Quantified Invariants via Predicate Abstraction," in *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, ser. LNCS, B. Stefen and G. Levi, Eds., vol. 2937. Springer-Verlag, 2004, pp. 267–281.

[12] J. S. Moore and G. Porter, "The Apprentice Challenge," *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, vol. 24, no. 3, pp. 1–24, May 2002.