The Integrated CWB-NC/PIOATool for Functional Verification and Performance Analysis of Concurrent Systems ⋆

Dezhuang Zhang, Rance Cleaveland, and Eugene W. Stark

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400 USA
{dezhuang, rance, stark}@cs.sunysb.edu

Abstract. This paper reports on an effort to integrate two verification tools, the Concurrency Workbench of the New Century (CWB-NC) and PIOATool. Our aim is to build a single tool that combines the "functional" analysis capabilities of the CWB-NC with the compositional performance-analysis features of PIOATool. We discuss some of the issues involved in the integration, highlighting a particular integration paradigm in which one tool becomes a *subshell* of the other.

1 Introduction

This paper describes a tool integration effort involving the combination of two system-analysis tools, the Concurrency Workbench of the New Century (CWB-NC) [4–6] and PIOATool [9, 12]. The goal of this project is to build a new tool combining support for checking both correctness and performance properties of system models.

The two tools in question have the following characteristics. The **CWB-NC** is a retargetable tool that implements a number of "functional correctness" routines, including a variety of semantic equivalences and preorders and a model checker for the modal μ-calculus. The **PIOATool** implements compositional performance analysis methods [9, 10] for *probabilistic I/O automata* (PIOA) [14]. PIOAs extend the well-known I/O automaton model for nondeterministic computation [8] with two kinds of performance information: probability distributions representing the relative likelihood with which transitions from a given state labeled by the same *input* are performed; and rate information describing how long, on average, outputs / internal actions take. PIOAs are also equipped with notion of parallel composition. The PIOATool computes a variety of transient performance measures on parallel compositions of PIOAs.

⋆ Research supported by NSF grants CCR-9988155, CCR-9988489, and CCR-0098037 and Army Research Office grants DAAD190110003 and DAAD190110019.

An important requirement for the integrated tool is that users should be able to provide system models on which both functional and performance analyses could be undertaken. However, the CWB-NC and PIOATool have very different internal model formats. The former generally supports compositional specification notations based on process algebras. Internally, systems are represented as terms in a modeling language; semantic routines are then used by the verification procedures such as the model checker in order to compute the semantic content of these terms. The PIOATool, on the other hand, includes procedures that are highly optimized for handling systems given as parallel compositions of semantic objects in the form of PIOAs.

We address these issues by: defining a process algebra for PIOA systems, implementing a translation for terms in this algebra into sequential compositions of "pure" PIOAs, and introducing a *subshell* for running PIOATool analyses. The rest of this paper discusses each of these in turn and concludes with a case study and related work.

2 A Process Algebra for PIOAs

The notation for PIOAs [11] has a two-level syntax. The lower level comprises so-called *sequential* terms, which denote transition systems annotated with probability and rate information. At this level, PIOA requirements such as "input-enabledness" (every state must be capable of processing any input) are not enforced. The upper level defines *PIOA agents*, which denote probabilistic I/O automata that satisfy all the input-enabledness, stochasticity, and compatibility properties that systems of PIOA must possess.

To define the language, let L be a set of labels, and let $\tau \notin L$ be the internal action; we use $a \in L$ and $b \in L \cup \tau$ in what follows. Let $p_1, \ldots, p_n$ denote probabilities summing to 1 and r a positive real number denoting a rate, and let X come from a set of *process names*. Then sequential terms are defined via the following BNF grammar.

$$s ::= nil \mid a?[p_1 : s_1, \ldots, p_n : s_n] \mid b(r)!s \mid s_1 + s_2 \mid X$$

Intuitively nil has no transitions, while $a?[p_1 : s_1, \ldots, p_n : s_n]$ can perform the input action $a?$ and subsequently evolve to term s_i with probability p_i, $b(r)!s$ denotes a process that can perform output/internal action $b!$ with rate r and then evolve to term s, and $s_1 + s_2$ is a nondeterministic choice. Finally, X represents an "invocation" of the process term bound to X in the environment.

The upper level of the language syntax is as follows, where $I \subseteq L$ and $O \subseteq L$ are sets of labels of input and output actions, respectively, and $a, a' \in L$.

$$t ::= \langle I, O \rangle s \mid [I, O]t \mid t_1 \parallel t_2 \mid t\{a \leftarrow a'\}$$

These operators have the following interpretation. $\langle I, O \rangle s$ represents a "type cast": provided every state reachable from s enables all, and only, inputs in I, and only outputs in O, and stochasticity requirements are met, then $\langle I, O \rangle s$ is a PIOA term. Term $[I, O]t$ denotes a "coercion" of t so that inputs come from I and outputs come from O. Term $t_1 \parallel t_2$ is the parallel composition of t_1 and t_2: for it to be well-formed, t_1 and t_2 must not share output actions. The resulting term has as inputs the intersection of the inputs of t_1 and t_2 and as outputs the union of the output sets of t_1 and t_2. Outputs of t_1 sharing a label with inputs of t_2 are "fed into" t_2 and also made available to the environment of $t_1 \parallel t_2$. Finally, $t\{a \leftarrow a'\}$ relabels the actions involving a to ones involving a'.

As the previous discussion implies, PIOA terms have a type system that ensures input-enabledness and compatibility of parallel compositions. The definition of this system, and of the SOS rules defining the transitions of PIOA terms, are omitted. The PAC [3] is applied to these rules to build the single-step "transition engine" used by the CWB-NC and PIOATool. Exhaustively applying the engine to a type-correct PIOA term yields a single PIOA describing the global behavior of the system.

3 Translating PIOA Terms to Parallel Compositions

The PIOA language fits easily within the CWB-NC framework: terms in the language represent systems, and the SOS rules define a transition relation. For the CWB-NC semantic analyses (which are insensitive to stochastic information) the probability and rate information on the transitions is ignored. However, for these terms to be processable by the PIOATool they must be converted into parallel compositions of "pure" PIOA. This entails the elimination of coercion and renaming and the replacement of sequential terms by PIOAs. The latter may be easily accomplished by applying the operational semantics exhaustively to these terms, yielding PIOAs. Coercion may also be eliminated, since in the PIOA language it turns out to distribute over parallel composition and may be "absorbed" easily into PIOAs. Somewhat surprisingly, the well-formedness conditions for PIOAs also license the distribution of renaming over parallel composition. Thus the basic conversion routine may be defined as follows. (1) "Push" all coercions and renamings inside all parallel compositions; (2) To the sequential + coercion + renaming terms embedded inside the parallel compositions, apply the SOS rules to generate PIOAs. Because parallel composition is associative, the resulting term containing PIOAs and parallel composition can be converted into a list of PIOAs. Note that no special semantic routines need to be implemented for this transformation; the existing PIOA semantic routines, and the algebraic properties of the language, suffice.

4 The PIOA Subshell

The final conceptual issue we confront involves the integration of the tool functionalities. The paradigm we adopt is based on the notion of a *subshell*, i.e. an "inner" command line, or *mode*, that has access to the data structures of the "outer" command line. In our case, we make the PIOATool a subshell of the CWB-NC. Once inside this subshell, the user may invoke the performance-analysis routines of PIOATool.

As was mentioned earlier, the two tools use different internal representations for systems. While there are routines for converting terms into sequences of PIOAs, these are computationally expensive. Consequently, we want to minimize the number of times these routines are called, while still providing a user with the analytical capabilities of both tools. Our solution to this issue is to introduce a separate environment for the PIOATool subshell. This environment maps identifiers to sequences of PIOAs. We also add a command to the subshell that allows PIOA terms to be "imported" into the PIOATool environment. This command translates the term given as an argument and binds it to an identifier. Once this importation takes place, the associated sequence of PIOAs may be subjected to PIOATool commands.

The PIOATool computes performance statistics for user-defined *observables*, which are rules for mapping PIOA execution sequences to numeric values. The PIOATool subshell also includes commands for binding observables to identifiers.

5 Case Study: A Distributed Mutual Exclusion Protocol

We now consider the application of the integrated tool to a "tournament-style" distributed mutual-exclusion protocol. Our example supposes a collection of *user* processes that are located at the leaves of a binary tree. Each user process requires from time to time the exclusive use of a resource, the allocation of which is managed by *arbiter* processes located at the interior nodes of the tree.

A user process requests the resource by performing a *request* output action, which synchronizes with a corresponding *request* input action of the user's parent arbiter. If the resource is currently held by the arbiter, and the arbiter has not already committed the resource to the user's sibling, then the arbiter will respond with a *grant* output, which synchronizes with a corresponding *grant* input to the user process. When the user is finished using the resource, it performs a *release* output to return the resource to its parent. The user must then wait for a *reset* input from the parent before it is permitted to make a new request.

If a user process makes a request and the resource is already committed to its sibling, then the arbiter ignores the request. When the sibling has finished with the resource and issued a *release* to the arbiter, then the subsequent *reset* action performed by the arbiter also resets the pending request, which must then be reissued by the user. If a user process requests the resource from its parent and the resource is not currently in the possession of any process in the subtree rooted at the parent, then the parent arbiter must request the resource from its own parent. This is done in exactly the same way as for a user process requesting the resource from its parent.

```
seq U_IDLE = request(1)!U_WAITING + reset?[1: U_IDLE] + grant?[1: U_ERROR]
seq U_WAITING = reset?[1: U_IDLE] + grant?[1: U_USING]
seq U_USING = release(1)!U_DONE + reset?[1: U_ERROR] + grant?[1: U_ERROR]
seq U_DONE = reset?[1: U_IDLE] + grant?[1: U_ERROR]
seq U_ERROR = error(1)!U_ERROR + reset?[1: U_ERROR] + grant?[1: U_ERROR]
proc USER = <{reset,grant},{request,release}>U_IDLE
```

The above shows the PIOA code for a user process. The code defines five sequential terms (introduced by the `seq` keyword), followed by the definition of PIOA term `USER`.

In the remainder of this section we step through a session in which we analyze a four-user system organized as a complete binary tree. This system has $1,264,375$ global states, of which $1,700$ are reachable. The session was run on a 1.8GHz Intel Xeon processor with 2Gb of on-board memory. Throughout, we quote the commands issued verbatim while simply summarizing the output produced by the tool.

```
cwb-nc> load mutex.pioa
cwb-nc> load mutex.mu
cwb-nc> chk MUTEX root_can_error
```

We begin the session by loading the file `mutex.pioa`, which contains the PIOA declarations needed to define `MUTEX`, the overall system. We then load file `mutex.mu`, which contains declarations of mu-calculus formulas defining various properties to be checked of `MUTEX`. One such property, `root_can_error`, asserts that the root arbiter is capable of entering an error state due to the arrival of an unexpected input (e.g. `release` before a `grant`). To check this property , we use the `chk` command, which invokes the CWB-NC's model checker. After 2.3 seconds, the CWB-NC responds with `FALSE`: the root arbiter cannot enter an error state.

```
cwb-nc> pioa
cwb-nc-pioa> sys S = MUTEX
cwb-nc-pioa> obs awaitRequest1 = await { R.SUBTREE[2].L.[].request_1 }
cwb-nc-pioa> obs awaitGrant1 = await { R.SUBTREE[2].L.[].grant_1 }
cwb-nc-pioa> obs req1ToGrant1Prob = (prob * (awaitRequest1 ; awaitGrant1))
cwb-nc-pioa> obs result = apply S req1ToGrant1Prob
cwb-nc-pioa> obs req1ToGrant1Time = (prob * (awaitRequest1 ; (time * awaitGrant1)))
cwb-nc-pioa> obs result = apply S req1ToGrant1Time
cwb-nc-pioa> eval result
```

The next part of the session shows the use of PIOATool to calculate two different quantities concerning the first request issued by the leftmost user in the system. The first such quantity is the measure ("probability") that this request will eventually be granted. To obtain this number, we first enter the PIOA subshell using the `pioa` command. Then, the PIOA term `MUTEX` is compiled into a system of PIOAs and bound to the identifier `S`. We next define an "observable" `request1ToGrant1Prob` using primitives `await`, `prob`, `;`, and `*` provided by PIOATool. The long string inside of the braces is an action pathname that is generated automatically by the tool to ensure that internal actions are given unique names. The PIOA system `S` is then "applied" to `request1ToGrant1Prob` to produce a new observable: `result`. The application of `S` actually proceeds in a component-at-a-time fashion. Finally `result` is "evaluated" to extract the resulting probability. This phase, which involves the solution of a system of linear equations in 537 unknowns, is performed using straight LU decomposition. The reported result, 1, is obtained after 149.1 seconds of computation, and represents the likelihood that the leftmost user does not "starve". This result is in contrast to what would be obtained a model checker, which would report that starvation is possible. The quantitative analysis indicates that the "likelihood" of starvation is 0.

Given that the leftmost user cannot, probabilistically speaking, starve, the next quantity we compute is the expected time that elapses between the user's `request` and the subsequent `grant`. To calculate this, we form the observable `req1ToGrant1Time`, apply `S` to it, and evaluate the result. After 72.2 seconds, the answer 15.85 is returned.

6 Conclusions, and Related and Future Work

In this paper we have studied issues surrounding the integration of a performance-analysis tool, PIOATool, and a functional-analysis tool, the Concurrency Workbench of the New Century. Central to our integration architecture is the notion of *subshell*, which allows the integrated tools to share data structures and routines while retaining their separate analytical capabilities. The subshell idea efficiently addresses the *data transfer* problem among integrating tools: how can the tools exchange information? A common approach involves the use of files to store this information. While this has the advantage of enabling tools to be loosely coupled, it does suffer from the following drawbacks. (a) An intermediate format must be defined. This can be subtle and time-consuming. (b) The intermediate formats must be parsed and unparsed repeatedly, slowing down tool performance. (c) The files can become quite large and require significant processing time when tools must exchange intermediate data structures.

The two existing efforts most closely related to this project include TwoTowers [2] and the performance integration work in CADP [7]. TwoTowers is also built on two existing tools, the CWB-NC and MarCA [13] with EMPAr [1] as the specification language. Aside from the specification formalisms, the key distinction between that work and this involves the tool architecture. TwoTowers isolates the CWB-NC and MarCA from one another; the tool kernel is responsible for translating EMPAr terms into a format suitable for either tool. The kernel therefore implements a semantics of (a subset of) EMPAr for the MarCA tool, and another semantics (embedded in the CWB-NC using PAC) for the functional interpretation of EMPAr. In contrast, our tool implements a single base semantics used by both tools. The CADP project uses files to transfer data between tools; our integration is tighter and therefore avoids overhead associated with accessing secondary storage to retrieve intermediate results.

As for future work, we would like to study how the CWB-NC/PIOATool can be modified to support performance analyses for value-passing systems.

References

1. M. Bernardo. An algebra-based method to associate rewards with EMPA terms. In *ICALP'97*, vol. 1256 of *LNCS*, pages 358–368, Jul. 1997.
2. M. Bernardo, R. Cleaveland, S. Sims, and W. Stewart. TwoTowers: A tool integrating functional and performance analysis of concurrent systems. In *FORTE XI/PSTV XVIII '98*, pages 457–467, Nov. 1998.

3. R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In *TACAS'95*, vol. 1019 of *LNCS*, pages 153–173, May 1995.
4. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM TOPLAS*, 15(1):36–72, Jan. 1993.
5. R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In *Computer Aided Verification (CAV '96)*, vol. 1102 of *LNCS*, pages 394–397, Jul. 1996.
6. R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, Jan. 2002.
7. H. Hermanns and H. Garavel. On combining functional verification and performance evaluation using CADP. In *FME*, vol. 2391 of *LNCS*, pages 410–429, 2002.
8. N. Lynch and M.Tuttle. Hierarchical correctness proofs for distributed algorithms. In *6th ACM PODC*, pages 137–151, 1987.
9. E. Stark and G. Pemmasani. Implementation of a compositional performance analysis algorithm for probabilistic I/O automata. In *7th PAPM*, pages 3–24, 1999.
10. E. Stark and S. Smolka. Compositional analysis of expected delays in networks of probabilistic I/O automata. In *Proc. 13th LICS*, pages 466–477, Jun. 1998.
11. E. Stark, S. Smolka, and R. Cleaveland. A process-algebraic language for PIOA. Unpublished draft, 2003.
12. E. Stark. Compositional performance analysis using probabilistic I/O automata. In *CONCUR 2000*, vol. 1877 of *LNCS*, pages 25–28, Aug. 2000.
13. W. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Presss, 1994.
14. S. Wu, S. Smolka, and E. Stark. Composition and behaviors of probabilistic I/O automata. *Theoretical Computer Science*, 176(1–2):1–38, Apr. 1997.