Scalable Software Model Checking Using Design for Verification*

Tevfik Bultan and Aysu Betin-Can

Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
{bultan,aysu}@cs.ucsb.edu

Abstract. There has been significant progress in automated verification techniques based on model checking. However, scalable software model checking remains a challenging problem. We believe that this problem can be addressed using a design for verification approach based on design patterns that facilitate scalable automated verification. We have been investigating a design for verification approach based on the following principles: 1) use of stateful, behavioral interfaces which isolate the behavior and enable modular verification, 2) an assume-guarantee style verification strategy which separates verification of the behavior from the verification of the conformance to the interface specifications, 3) a general model checking technique for interface verification, and 4) domain specific and specialized verification techniques for behavior verification. So far we have applied this approach to verification of synchronization operations in concurrent programs and to verification of interactions among multiple peers in composite web services. The case studies we conducted indicate that scalable software verification is achievable in these application domains using our design for verification approach.

1 Introduction

Automated software verification techniques based on model checking have improved significantly in recent years. When combined with the increasing computing power, these techniques are capable of analyzing complex software systems as demonstrated by numerous case studies. However, most applications of software model checking succeed either by requiring some manual intervention or by focusing on a specific type of software or a specific type of problem. It is still unclear if there is a general framework for scalable software model checking.

Scalability of software model checking depends on extracting compact models from programs that hide the details that are not relevant to the properties being verified. This typically requires a reverse engineering step in which either user guidance or static analysis techniques (or both) are used to rediscover some information about the software that may be known to its developers at design time. A design for verification approach, which enables software developers to document the design decisions that can be useful

* This work is supported by NSF grant CCR-0341365.

for verification, may improve the scalability and therefore the applicability of model checking techniques significantly.

We have been investigating the use of design for verification for different applications of software model checking. We believe that it is possible to develop a general design for verification approach based on stateful, behavioral interfaces which can be used to decouple behavior of a module from its environment. This decoupling enables an assume-guarantee style modular verification strategy which separates verification of the behavior from the verification of the conformance to the interface specifications. Interface verification can be performed using generic software model checking techniques whereas behavior verification can be performed using domain specific and specialized verification techniques. The modularity and the specialization of the verification tasks are crucial for the scalability of our approach. We proposed a set of design patterns which facilitate modular specification of interfaces and behaviors. These design patterns also help us in automating the model extraction and environment generation steps required for model checking.

So far we have applied our design for verification approach to verification of synchronization operations in concurrent programs [1, 2, 5, 25] and to verification of interactions among multiple peers in composite web services [3, 4]. We believe that this approach can be extended to a general framework in which software developers write behavioral interfaces while they are building different modules, and these interfaces are used by the model checking tools to achieve scalable software verification.

Related Work. Earlier work on design for verification focused on verification of UML models [19] and use of design patterns in improving the efficiency of automated verification techniques [17]. There has been some work on behavioral interfaces in which interfaces of software modules are specified as a set of constraints, and algorithms for interface compatibility checking are developed [8]. Also, there has been work on extending type systems with stateful interfaces [9], suggesting an approach in which interface checking is treated as a part of type checking. Assume guarantee style verification of software components has also been studied [18] in which LTL formulas are used to specify the environment (i.e., the interface) of a component. Automated environment generation for software components has been investigated using techniques such as inserting nondeterminism into the code and eliminating or restricting the input arguments by using side effect and points-to analyses [14, 20, 21]. Finally, ESC Java [11] uses an approach based on design by contract and automated theorem proving which is similar to what we are proposing here for model checking.

Below, we will first discuss interfaces and modularity and interface-based verification in general terms. Later, we will discuss application of these principles to verification of synchronization operations in concurrent programs and to verification of interactions among multiple peers in composite web services. We will end the paper with a brief discussion of what needs to be done to generalize our approach.

2 Interfaces, Modularity and Interface-Based Verification

Modularization of the verification task is necessary for its scalability. Modularization requires specification (or discovery) of the module interfaces. In order to achieve mod-

ularity in verification, the module interfaces have to provide just the right amount of information. If the interfaces provide too much information, then they are not helpful in achieving modularity in verification. On the other hand, if they provide too little information, then they are not helpful for verifying interesting properties. Interface of a module should provide the necessary information about how to interact with that module without giving all the details of its internal structure.

Current programming languages do not provide adequate mechanisms for representing module interfaces because they provide too little information. Think of an object class in an object oriented language. The interface of an object class consists of names and types of its fields, and names, return and argument types of its methods. Such interface specifications do not contain sufficient information for most verification tasks. For example, such an interface does not contain any information about the order the methods of the class should be called. In order to achieve modular verification, module interfaces need to be richer than the ones provided by the existing programming languages.

We believe that finite state machines provide an appropriate tool for specification of module interfaces. Such interface machines can be used to specify the order of method calls or any other information that is necessary to interact with a module. As an example, consider the verification of a concurrent bounded buffer implementation. Access to the buffer operations can be protected with user defined synchronization operations. For example, one requirement could be that the synchronization method *read-enter* should be called before the *read* method for the buffer is called. Such constraints can be specified using an interface machine which defines the required ordering for the method calls. For complex interfaces one could use an extended state machine model and provide information about some of the input and output parameters in a module's interface. Another possible extension is to use hierarchical state machines for interfaces [3].

Behavioral interfaces enable an assume-guarantee style verification strategy which separates behavior and interface verification steps. Interfaces enable isolation of the behavior of interest by separating it from its environment. The behavior of interest could be the behavior of an object encapsulated in an object class or it could be the interaction among multiple components in a distributed system. A behavioral interface for the object class can be used to isolate the object behavior by decoupling it from its environment. Similarly, interfaces of different components can be used to isolate the interaction among multiple components from the component implementations.

Behavior Verification: In the proposed interface-based verification approach, during behavior verification it is assumed that there are no interface violations in the software. Based on this assumption, interfaces are used as environment models. Environment generation is a crucial problem in software model checking [14, 21]. We are suggesting the use of a design for verification approach to attack this problem. Software developers are required to write interfaces during the software development process so that these interfaces can then be used as a model of the environment during behavior verification. Using such interfaces we can encapsulate the behavior in question and perform the verification on this encapsulated behavior separately. Note that, interfaces should represent all the constraints about the environment that are relevant to the behavior of interest, i.e., the interfaces should provide all the information about the environment that is nec-

essary to verify the behavior. This is analogous to requiring programmers to declare types to enable type checking.

During behavior verification one could use domain specific verification techniques. Recall the concurrent buffer example above and assume that this buffer is implemented as a linked list. During behavior verification we can assume that the threads that access to this buffer obey its interface and we can verify the correctness of the linked list implementation without worrying about the interface violations. For the verification of the linked list we can use specialized verification techniques such as shape analysis [23] or infinite state model checking [6, 24]. Since, interfaces allow isolation of the behavior, application of domain specific verification techniques (which may not be applicable or scalable in general) becomes feasible. These domain specific verification techniques may enable verification of stronger and more complex properties than that can be achieved by more generic techniques.

Interface Verification: During interface verification we need to verify that there are no interface violations. If interfaces of different modules are specified uniformly, for example using finite state machines, this step can be handled using a uniform verification technique. Note that, this verification technique should be capable of handling different types of modules. Hence, during interface verification, it is more suitable to use the generic verification techniques developed for the purpose of applying model checking directly to existing programming languages [13, 22].

During interface verification, the interfaces can be used to abstract the behavior that is verified during behavior verification. Recall the concurrent buffer example. Interface verification step for this example requires that each thread which has access to the concurrent buffer has to be checked for interface violations. During the verification of a thread, the behavior of the concurrent buffer can be abstracted by replacing the concurrent buffer by its interface machine. Note that, here, we are assuming that the concurrent buffer itself does not make calls to its methods directly or indirectly. If that is not the case, the behavior of the concurrent buffer and any other part of the code which is not relevant to the interface violations can be abstracted away using static analysis techniques such as slicing, since we are only interested in interface violations.

3 Applications

So far we have applied the approach discussed above to verification of synchronization operations in Java programs and to verification of interactions among multiple peers participating to a web service implemented in Java.

3.1 Application to Concurrent Programming

We applied the above principles in developing a design for verification approach for concurrent programming in Java with the goal of eliminating synchronization errors from Java programs using model checking techniques [1, 2, 25]. We developed a design pattern, called concurrency controller pattern, in which synchronization policies

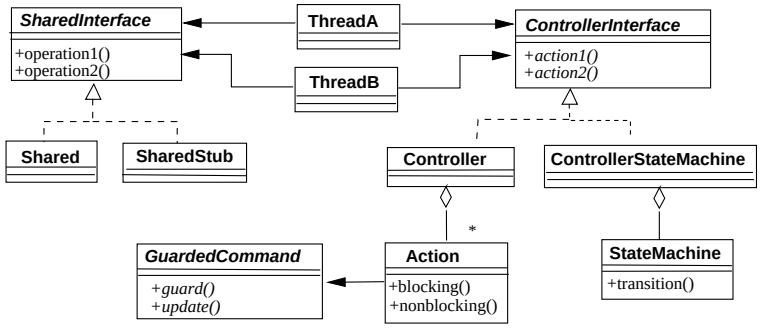


Fig. 1. Concurrency Controller Pattern Class Diagram

for coordinating the interactions among multiple threads are specified using concurrency controller classes. The behavior of a concurrency controller is specified as a set of actions (forming the methods of the controller class) where each action consists of a set of guarded commands. The controller interface is specified as a finite state machine which defines the order that the actions of the controller can be executed by each thread.

Figure 1 shows the class diagram for the concurrency controller pattern. The `ControllerInterface` is a Java interface which defines the names of the controller actions. The `Controller` class contains the actions specifying the controller behavior. The `Action` class is the helper class containing a set of guarded commands and implements the semantics of action execution. This class is provided with the concurrency controller pattern, i.e., the developers do not need to modify it. Same holds for the `GuardedCommand` Java interface. The `ControllerStateMachine` class is the controller interface. This class has an instance of the `StateMachine` which is a finite state machine implementation provided with the pattern and can be used as is. The `SharedInterface` is the Java interface for the shared data. The actual implementation of the shared data is the `Shared` class. The class `SharedStub` specifies the constraints on accessing the shared data based on the interface states of the controller.

Recall the concurrent buffer example. We can coordinate the concurrent accesses to this buffer with a controller class which implements a bounded buffer synchronization protected by a reader-writer lock. The synchronization strategy implemented by this controller will allow multiple threads to read the contents of the buffer at the same time but it will only allow a thread to insert or remove an item from the buffer when there is no other thread accessing the buffer. Additionally, this concurrency controller will ensure that a thread that wants to insert an item to the buffer will wait while the buffer is full. Similarly, a thread that wants to remove an item from the buffer will wait while the buffer is empty. We call the concurrency controller which implements this synchronization BB-RW.

The BB-RW controller can be implemented using four variables and five actions. The variables are `nR` denoting the number of readers in the critical section, `busy` denoting if there is a writer in the critical section, `count` denoting the number of items in the buffer, and `size` denoting the size of the buffer. The actions are `r_enter`, `r_exit`,

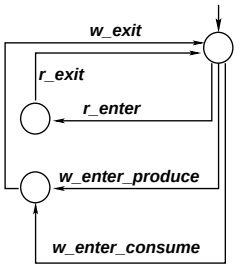


Fig. 2. Interface of BB-RW

```
class BBRWController implements BBRWInterface{
 int nR; boolean busy; int count; final int size;
 r_enter=new GuardedCommand() {
   public boolean guard() {return (!busy);}
   public void update() {nR = nR+1;}};
 r_exit=new GuardedCommand() {
   public boolean guard() {return true;}
   public void update() {nR = nR-1;}};
 w_enter_produce=new GuardedCommand() {
   public boolean guard() {
      return (nR == 0 && !busy && count<size);}
   public void update() {
      busy = true; count=count+1;}};
 w_exit_consume=new GuardedCommand() {
   public boolean guard() {
      return (nR == 0 && !busy && count>0); }
   public void update() {
      busy = true; count=count-1;}};
 w_exit= new GuardedCommand() {
   public boolean guard() { return true; }
   public void update() { busy = false; } };
 ...
}
```

Fig. 3. BB-RW Controller Implementation

`w_enter_produce`, `w_enter_consume`, and `w_exit`. A part of the BB-RW controller implementation is given in Figure 3. The interface of the BB-RW controller is shown in Figure 2 where the transitions of the interface machine are labeled with the actions of the BB-RW controller.

We developed a modular verification strategy based on the concurrency controller pattern. We first verify automatically generated infinite state models of concurrency controllers using the symbolic and infinite state model checker Action Language Verifier [24], assuming that the threads using the controllers obey their interfaces. Next, we verify this assumption using the explicit and finite state model checker Java PathFinder [22]. In this modular verification strategy the two verification steps are completely decoupled. Moreover, during the verification of the threads for interface violations there is no need to consider interleavings of different threads since we are only interested in the order of calls to the controller methods by each individual thread, and since the only interaction among different threads is through shared objects that are protected using the concurrency controllers, i.e., we can verify each thread in isolation [5].

We conducted two case studies to demonstrate the effectiveness of this approach. The first case study was a concurrent editor which was implemented with 2,800 lines of Java code using a client-server architecture [2]. The concurrent editor allows multiple users to edit a document concurrently as long as they are editing different paragraphs and maintains a consistent view of the shared document among the client nodes and the server. In this case study there were 4 mutex controller instances, one reader-writer controller per paragraph, one bounded buffer (with mutex lock) controller per paragraph, and one barrier controller. The concurrent editor had 4 threads in the client node and 2 threads in the server node. The second case study [5] was conducted on a safety critical air traffic control software called Tactical Separation Assisted Flight Environment (TSAFE) [10]. We reengineered the distributed client-server version of TSAFE which consists of 21,000 lines of Java code. The server node stores flight trajectories in

a database, receives current flight data from a radar feed through a network connection to update the database, and monitors the conformance of the flights to their trajectories. The client nodes display the flight status information. The reengineered system used 2 reader-writer controller instances and 3 mutex controller instances. TSAFE had 3 threads in the client node and 4 threads in the server node. In both of these case studies, the behavior verification of the controllers took less than a few seconds and used less than 11 MB memory. In these case studies we isolated the threads for interface verification with automatically synthesized drivers and stubs, i.e., the interface verification was performed thread modularly. Interface verification for some threads took several hundreds seconds and for some of them it took a few dozen seconds. The longest interface verification time we recorded was 1636.62 seconds. The maximum memory consumption recorded during interface verification was less than 140 MB.

3.2 Application to Web Services

We also developed a design for verification approach for verification of web services based on the above principles [3, 4]. We focused on composite web services which consist of asynchronously communicating peers. Our goal was to automatically verify properties of interactions among such peers. We modeled such interactions as conversations, the global sequence of messages that are exchanged among the peers [7]. We proposed a design pattern for the development of such web services which enables a modular, assume-guarantee style verification strategy. In the proposed design pattern, called peer controller pattern, each peer is associated with a behavioral interface description which specifies how that peer will interact with other peers. Assuming that the participating peers behave according to their interfaces, we verify safety and liveness properties about the global behavior of the composite web service during behavior verification. During interface verification, we check that each peer implementation conforms to its interface. Using the modularity in the proposed design pattern, we were able to perform the interface verification of each peer and the behavior verification as separate steps. Our experiments showed that, using this modular approach, one can automatically and efficiently verify web service implementations.

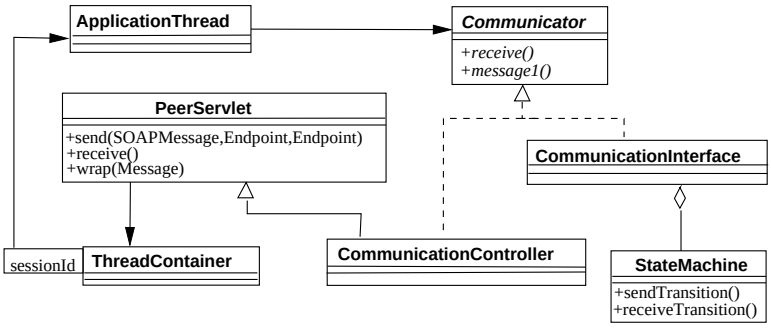
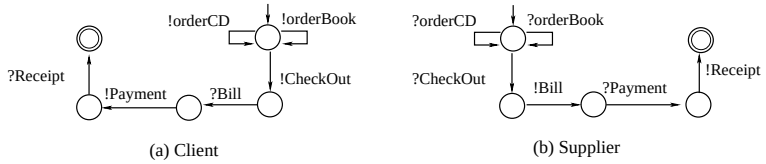


Fig. 4. Peer Controller Pattern Class Diagram

The class diagram of the peer controller pattern is shown in Figure 4. The application logic is implemented with the `ApplicationThread`. Each instance of this thread is identified with a session number. The application thread communicates asynchronously with other peers through the `Communicator` which is a Java interface that provides standardized access to the communication implementation. The `CommunicationController` class is a servlet that performs the actual communication. Since it is tedious to write such a class, we provide a servlet implementation (`PeerServlet`) that uses JAXM [16] in asynchronous mode. The `PeerServlet` is associated with a `ThreadContainer` which contains application thread references indexed by session numbers. When a message with an associated session number is received from the JAXM provider, it is delegated to the thread indexed with that session number. The behavioral interface of each peer is written as an instance of the `CommunicatorInterface` class and contains a finite state machine specification written using the provided `StateMachine` helper class.



(a) Client (b) Supplier

Fig. 5. Client-Supplier Example

Consider a composite web service with two peers: one client and one supplier. The client peer places arbitrary number of CD and book orders. After ordering the products, the client issues a *CheckOut* message. The supplier calculates the total price and sends a bill to the client. Client sends the payment and gets a receipt from the supplier. The state machines defining the contracts of these peers are shown in Figure 5. We verified the behavior of this example with different queue sizes using the Spin model checker [15]. Note that, using the behavioral interfaces in the peer controller pattern, we can easily extract the behavior specification characterizing the interactions of a web service composition. For the example shown in 5, the state space of the behavior specification increases exponentially with the size of the queues. In fact, the number of reachable states for this example is infinite if unbounded queues are used. The exponential growth in the state space affects the performance of the Spin model checker significantly. In fact, Spin ran out of memory when the queue size was set to 15.

We adapted the synchronizability analysis [12] into our framework in order to verify properties of composite web services in the presence of asynchronous communication with unbounded queues. A composite web service is called synchronizable if its global interaction behavior (i.e., the set of conversations) does not change when asynchronous communication is replaced with synchronous communication [12]. The synchronizability analysis enables us to reason about global behaviors of composite web services with respect to unbounded queues. It also improves the efficiency of the behavior verification by removing the message queues, which reduces the state space. Our automated syn-

chronizability analyzer identified the client-supplier example discussed above as synchronizable. With synchronous communication the reachable state space contained only 68 states and the behavior verification succeeded in less than 0.01 seconds using less than 1.5 MB of memory.

We applied the above design for verification approach to a loan approval system with three peers [4], a travel agency system with five peers, and an order handling system with five peers [3]. We used hierarchical finite state machines for specifying the peer interfaces of the latter two examples. The behavior verification for all these examples took a few seconds. Using the automated synchronizability analysis, we identified that all of these examples are synchronizable. This result lead to improvements in the behavior verification since synchronous communication results in less number of states. The travel agency system mentioned above had an infinite set of reachable states. Using synchronizability analysis, we were able to verify the global behavior of this example for unbounded message queues. During the verification of the peer implementations, we used the peer interfaces to isolate the peers. This caused a significant reduction in the state space which improved the performance of the interface verification. The interface verification for any of the peers in these examples took a few seconds (at most 9.72 seconds) and did not consume a significant amount of memory (at most 19.69 MB).

4 Conclusions

Our experience in design for verification of concurrent and distributed software systems leads us to believe that software model checking can become a scalable verification technique as long as the software developers write behavioral interfaces which can be exploited for modular verification. In the application domains discussed above, we implemented this approach by providing design patterns that enable specification of behavioral interfaces in existing programming languages. Alternatively, one can extend the programming languages with primitives that allow the specification of behavioral interfaces. Either way, for scalability of model checking, we need to investigate more ways of collecting information about the structure of software during the design phase rather then reverse engineering programs to discover their structure during verification.

There are numerous challenges to be addressed in the behavior and interface verification steps discussed above. We need a large set of domain specific verification techniques to handle different types of behavior verification. The interface-based verification approach discussed above can be used to integrate a diverse set of verification techniques under a single framework. One of the biggest challenges in the presented approach is the development of a uniform interface verification technique. Although this is a challenging problem it is less challenging than the general software model checking problem since it only focuses on interface violations. Automated abstraction techniques may be used more effectively to exploit this focus, and, when combined with a modular verification strategy, this can lead to scalable verification.

References

1. A. Betin-Can and T. Bultan. Interface-based specification and verification of concurrency controllers. In *Proc. SoftMC, ENTCS*, volume 89, 2003.

2. A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proc. 19th IEEE Int. Conf. on ASE*, pages 248–257, 2004.
3. A. Betin-Can and T. Bultan. Verifiable web services with hierarchichal interfaces. In *Proc. IEEE Int. Conf. on Web Services*, pages 85–94, 2005.
4. A. Betin-Can, T. Bultan, and X. Fu. Design for verification for asynchronously communicating web services. In *Proc. 14th WWW Conf.*, pages 750–759, 2005.
5. A. Betin-Can, T. Bultan, M. Lindvall, S. Topp, and B. Lux. Application of design for verification with concurrency controllers to air traffic control software. In *Proc. 20th IEEE Int. Conf. on ASE (to appear)*, 2005.
6. B. Boigelot, P. Godefroid, B. Williams, and P. Wolper. The power of QDDs. In *Proc. 4th Static Analysis Symp.*, pages 172–186, 1997.
7. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. 12th WWW Conf.*, pages 403–410, 2003.
8. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdziński, and F.Y.C. Mang. Interface compatibility checking for software modules. In *Proc. CAV*, pages 428–441, 2002.
9. R. DeLine and M. Fahndrich. Typestates for objects. In *Proc. ECOOP*, pages 465–490, 2004.
10. G. Dennis. TSAFE: Building a trusted computing base for air traffic control software, Master's Thesis, 2003.
11. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proc. POPL*, pages 234–245, 2002.
12. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. 13th WWW Conf.*, pages 621–630, 2004.
13. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. POPL*, pages 174–186, January 1997.
14. P. Godefroid, C. Colby, and L. Jagadeesan. Automatically closing open reactive programs. In *Proc. PLDI*, pages 345–357, June 1998.
15. G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Eng.*, 23(5):279–295, May 1997.
16. Java API for XML messaging (JAXM). http://java.sun.com/ xml/jaxm/.
17. P. C. Mehlitz and J. Penix. Design for verification using design patterns to build reliable systems. In *Proc. Work. on Component-Based Soft. Eng.*, 2003.
18. C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume guarantee model checking of software: A comparative case study. In *Proc. Spin Workshop*, pages 168–183.
19. N. Sharygina, J. C. Browne, and R. P. Kurshan. A formal object-oriented analysis for software reliability: Design for verification. In *Proc. FASE*, pages 318–332, 2001.
20. O. Tkachuk and M. B. Dwyer. Adapting side-effects analysis for modular program model checking. In *Proc. ASE*, pages 116–129, 2003.
21. O. Tkachuk, M. B. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proc. ESEC/FSE*, pages 188–197, 2003.
22. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
23. R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proc. 9th Int. Conf. on Compiler Construction*, pages 1–17, 2000.
24. T. Yavuz-Kahveci, C. Bartzis, and T. Bultan. Action language verifier, extended. In *Proc. CAV*, pages 413–417, 2005.
25. T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proc. ISSTA*, pages 169–179, 2002.