Automatic Verification of Annotated Code*

Doron Peled and Hongyang Qu

Department of Computer Science, The University of Warwick,
Coventry CV4 7AL, UK

Abstract. Model checking is an automatic approach for the verification of systems. Explicit states model checking applies a search algorithm (e.g., depth or breadth first search) to the state space of the verified system. In concurrent systems, and in particular in communication protocols, the number of states can grow exponentially with the number of independent components (processes). There are many different methods that attempt to automatically reduce the number of checked states. Such methods show encouraging results, but often still fail to reduce the number of states required for the verification to become manageable. We propose here the use of code annotation in order to control the verification process and reduce the number of states searched. Our extension of the C programming language allows the user to put into the code instructions that are executed by the model checker during the verification. With the new language construct, we may exploit additional insight that the verifier may have about the checked program in order to limit the search. We describe our implementation and present some experimental results.

1 Introduction

Model checking [3] and testing [19] stand for two different ends of formal methods. The former tends to be more comprehensive in finding errors, while the latter only samples the execution, but is sometimes more affordable. There is a clear tradeoff between these two approaches, and there have been many attempts to optimize both the coverage and the amount of time and memory required.

We present here an approach that allows fine-tuning the verification search. The approach is based on adding annotations to the verification code. The annotations change the behavior of the checked code in a controlled way, allowing the user to draw the right conclusions from the verified results. The annotations allow avoiding part of the verification or testing effort performed by the search engine. It may not cover some of the cases, hence compromise the exhaustiveness of the verification for the sake of practicality. Yet the annotation does not create false negative error traces. This helps making affordable choices in the spectrum between the more comprehensive model checking and the more economic testing.

* This research was partially supported by Subcontract UTA03-031 to The University of Warwick under University of Texas at Austin's prime National Science Foundation Grant #CCR-0205483.

H. König, M. Heiner, and A. Wolisz (Eds.): FORTE 2003, LNCS 2767, pp. 127–143, 2003.

The idea of adding code to the checked software is already used in deductive program verification [1, 9]. It is exploited there to obtain information that does not exist when observing only the program variables. In many cases, annotations are merely used to explicitly represent the program counters. There, one is allowed to add new variables, often attributed as *history* variables. Adding assignments to these variables, based on the values of the program variables, is allowed. The additional code, i.e., the annotations, cannot change the behavior of the program with respect to the original program variables. Simple annotations are also in frequent use when testing software. A common use is the addition of invariants or simple conditions that are checked for violations during testing or run time, see, e.g., [7].

The tool ESC/Java [8] takes assertions annotating the code of a Java program written inside comments. Such assertions typically refer to type correctness problems, such as 'array indexes out of range', or 'division by zero'. The tool propagates these assertions backwards using the weakest precondition [5] predicate transformer. It then uses automatic procedures for theorem proving in order to show that the annotated properties hold. Due to the inherent undecidability of program verification, such a verification method is necessarily incomplete. On the other hand, unlike in model checking, this analysis does not need to conform with the finiteness restrictions of model checking.

In this paper, we suggest new programming constructs for representing program annotation, which can help in controlling the validation search. The annotations can include additional variables that are updated during the systematic search of the state space of the program. The added code controls the search and allows reducing the time and memory needed. On the other hand, it may reduce the exhaustiveness of the tests or verification effort. Care is taken to permit only annotations that do not cause incorrect conclusions about the correctness of the software or affect the viability of the discovered counterexamples.

The annotations play a different role than the original code. They are not part of the verified code and can be seen as a way to program the model checker in order to specialize the search. They are used for obtaining information during the model checking, which is collected using new variables, disjoint from the original program variables. These new variables can be used and be updated only within the annotations. On the other hand, the annotations can use the values of the program variables in updating these new variables. The state space search can then be controled using the gathered information. They can force immediate backtracking, commit to the nondeterministic choices made so far or terminate the search and report the current execution.

Model checking may fails because of time or space restrictions. Many methods are suggested to combat this problem (see [3] for an extensive survey of model checking techniques). These methods are effective in different (but not necessarily disjoint) instances. Experience shows that a verification engineer can use some limited access to the search control in order to exploit additional knowledge about the verified code. Such access is seldom provided by verification tools. We do not allow the user an open architecture model checking system. Instead, we

provide a safe and limited mechanism for controlling the testing or verification search.

Annotations are also useful in test case generation. In software testing [19] it was realized that it is pointless to try and explore all the executions of a program. Instead, one exploits some coverage criterion to derive a set of representative executions. Such representatives may cover, for example, the statements, the conditions or the data flow paths of the code [19]. If we naively allow a search engine to cover the state space of a system until we run out of time or memory, we will typically end up having very poor coverage by any reasonable criterion. Using annotations allows us to space-out the visited executions of the checked software, achieving better control of the coverage.

We were informed by one of the reviewers for this paper that a related method was developed in parallel by Gerard Holzmann for his new version of Spin (to be described in the new reference manual [11]). The new Spin version is capable of annotating the checked code using C commands, including adding new variables to support the annotation. One difference between the Spin annotation and ours is that we allow structural annotation, namely, annotating programming structures, e.g., as all the actions associated with a `while` loop, and the nesting of annotations. We also separate in the annotations the enabledness condition from the code to be applied. Our annotation methodology is strongly based upon this feature, allowing us to control the search by making some search directions disabled through the annotations.

2 Preliminaries

Explicit state automatic verification techniques [24] are usually based on a search through the state space of the analyzed system. The system is translated first into a collection of *atomic actions* [17, 21] (or simply *actions*). Atomic actions are the smallest visible units that can be observed to induce a change in the system. A *state* represents the memory of the system at some point of the computation. It is often described as a mapping from the program variables, the interprocess message queue and the program counters into their values. An action contains two parts: a *condition*, and a *transformation*. An action is *enabled* at a state, if its condition holds for that state. Then it can be executed, in which case, the transformation is applied to the state. By applying the transformation of an enabled action to a state we obtain a new state, usually causing at least one of the program counters value to be changed. Concurrent systems often allow *nondeterminism*, when there is a choice of more than a single atomic action enabled from some states. Nevertheless, each action is itself deterministic, i.e., when applied to the same state it will always generate the same successor. Some states are distinguished as the *initial states* of the system.

An *execution* is a finite or infinite alternating sequence of states and actions $s_0\,\alpha_0\,s_1\,\alpha_1\,\ldots$ where (1) s_0 is an initial state, (2) α_i is enabled at the state s_i, for $i \geq 0$ and (3) s_{i+1} is obtained from s_i by applying the transformation of α_i, for $i \geq 0$. We denote the set of executions of a system A by $\mathcal{L}(A)$ (the *language* of A).

Depending on the specification formalism used (e.g., linear temporal logic [22]), we may decide to include in $\mathcal{L}(A)$ only the sequences of states or the sequences of actions, projecting out the actions or states components, respectively. The state space of a system is a graph $\langle S, E\rangle$, where the nodes S represent the states and the directed edges E are labeled by actions, such that $s \xrightarrow{\alpha} s'$ when $s' \in S$ is obtained from $s \in S$ by applying α. Let $I \subseteq S$ be the set of initial states. An execution is hence represented as a path in the graph, starting from an initial state. A state is *reachable* if it appears in some execution of the system. A search through the state space of a system can be performed in order to exercise the code for testing, to check violations of some invariants, or to compare the collection of execution sequences with some system specification.

For a concurrent system with multiple parallel processes, we obtain the collection of all the actions of the various processes. An execution is still defined in the same way, allowing actions from different processes to interleave in order to form an execution. In some cases, we may want to impose some *fairness* constraints [10], disallowing an execution where, e.g., one process or one transition is ignored from some state continuously (i.e., in every state) or infinitely often. Note that we did not impose maximality or fairness on sequences, as our implementation, at this point, concentrates on checking safety.

A search of the state space has several distinct parameters:

- The direction of the search. A *forward* search often starts from the initial states and applies actions to states, obtaining their successors [11]. Applying an action to a reachable state guarantees obtaining a reachable state. A *backward* search is applied in the reverse direction, and does not necessarily preserves reachability [18].
- *Explicit* [11, 14] or *symbolic* search [11]. In an explicit search, we usually visit the states one at a time, and represent them individually in memory. In symbolic search we represent a collection of states, e.g., using some data structure or a formula. By applying an atomic action, we obtain a representation of a new collection of states (either the successors or the predecessors of the previous collection).
- The search strategy. Algorithms such as *depth first search* (DFS) [11] or *breadth first search* (BFS) [14] can be used. Different search strategies have distinct advantages and disadvantages. For example, BFS can be used to obtain a shortest execution that violates some given specification. On the other hand, DFS can focus more efficiently on generating the counterexample.
- The reduction method. Because of the high complexity of the search, we often apply some techniques that reduce the number of states that we need to explore. Reduction methods are based on certain observations about the nature of the checked system, e.g., commutativity between concurrent actions, symmetry in the structure of the system [4], and data independence [15, 23]. The goal of the reduction is to enhance the efficiency of the search and be able to check bigger instances, while preserving the correctness of the analysis.

In this paper we concentrate on forward explicit search using DFS. Our framework can be applied to BFS or to heuristic search [16]. However, in the case of a search other than DFS, the search order is different, which affects the way our method would work. The focus in this paper is on controlling the search with annotations for reducing the amount of time and memory required for the search. This can allow an automatic analysis of the system even when the size of the state space is prohibitively high for performing a comprehensive search. In particular, the explicit search strategy often becomes hopelessly large quite quickly. Consider for example a software system where three independent short integer values (holding 16 bits each) are used. This gives a combination of 2^{48} bits. Reduction and abstraction methods may help in some cases, but do not provide a solution that uniformally works. Symbolic model checking does not visit and represent each state individually, hence can sometimes be applied to cases that seem 'hopeless' for explicit states model checking [6]. Nevertheless, there are several cases where one prefers the explicit search. This is in particular useful for testing or simulating the code, and is also simpler for generating counterexamples.

3 Verification of Annotated Code

We allow annotating the checked program with additional code that is applied during the automatic verification process. The suggested syntax of the annotations is presented later in this section. Our syntax is given for C programs, but similar syntax can be formed for other programming languages. We allow two ways of adding annotations to the code. The first way is to put the annotation in between other program segments (for example, after an `if` statement, and before a `while` loop). An annotation is effective during verification when program control passes it, but not when it skips over it, e.g., using a `goto` statement.

A second way to annotate the program is to associate an annotation with a construct such as a `while` or an `if` statement. In this case, the effect of the annotation is imposed on each action translated from this construct. For example, if we annotate a `while` statement, the annotation will be effective with actions calculating the `while` condition (which may be done in one or more atomic actions) and actions that correspond to the `while` loop body.

In order to understand how the annotations work, we need to recall that before verifying a program, it is translated into a set of atomic actions. The annotations are also translated into actions, called *wrap actions*. The granularity of the former kind of actions is important in modeling the program since the model can behave differently with different granularities (see, e.g., [2]). Consequently, we often apply restrictions such as not allowing an atomic action to define (change) or use (check) more than a single shared program variable [20]. The issue of atomicity does not apply to the wrap actions since they do not represent the code of the checked program. Wrap actions can exploit the full flavor of a sequential part of the original programming language used, including several instructions and in particular using and defining multiple variables.

We allow nesting of annotations. Therefore, some code can be enclosed within multiple annotations. This means that an atomic action can be related to multiple wrap actions. When processing (executing) an atomic action, the model checker also processes all the wrap actions related to it. Both the atomic and the wrap actions may have a condition and a transformation. Applying a combination of an atomic action and wrap actions to a state requires that the conjunction of *all* their conditions hold in that state. This means that annotating a program may have the effect of blocking some actions that could be executed before annotating. This provision may be used to disallow some unuseful directions in the search, or to compromize exhaustiveness for practical purposes. This needs to be done carefully, as it can also render the search less exhaustive. When enabled, the transformation part of the atomic action together with all the corresponding wrap actions are executed according to some order, e.g., 'deeper nested actions are executed later'.

The transformation of wrap actions can be an arbitrary code. This may include iterative code, such as a `while` statement. It is the responsibility of the person adding the annotation to take care that they are not a source of infinite loops. We could have enforced some syntactic restrictions on annotations, but iterative constructs inside annotations seem to be useful.

3.1 Programming Constructs for Annotations

Programming languages are equipped with a collection of constructs that allow them to be effective (Turing complete). Adding a new construct to a sequential programming language means adding extra convenience rather than additional expressiveness. In concurrent programming languages, additional constructs may also introduce new ways of interaction between the concurrent agents.

We suggest here programming constructs that can help with the program testing or verification search. The verified code itself is often simulated, step by step, by the testing or verification engine. This simulation usually includes saving the current state description, and providing reference points for future backtracking. The additional code annotates the program in a way that allows controlling the search.

Special Variables

Before presenting the new programming constructs, we introduce two new types of variables that can be used with the annotation. We will call the original variables that occur in the verified code, including any variable required to model the behavior of the program (such as program counters, message queues) *program variables*.

History Variables. These variables are added to the state of the program. Their value can depend on the program variables in the *current checked execution*. Because actions are deterministic, the value of a history variable in a state s_i in an execution $s_0\alpha_0 s_1\alpha_1 \ldots s_i\alpha_i \ldots$ is a function of the initial state $s_0 \in I$ and

the sequence of actions $\alpha_0\alpha_1 \ldots \alpha_{i-1}$. Some examples of uses of history variables include:

- Limiting the number of times some statement can be executed in the currently checked execution prefix.
- Witnessing that some state property held in some state of the current execution.

Updating the history variables based on values of the program variables is allowed, but not vice versa. When backtracking, the value of the history variables return to their previous value.

Auxiliary Variables. These variables are updated while the search is performed but are not part of the search space. Thus, unlike history variables, their value is not rolled back when backtracking is performed. Examples for uses of auxiliary variables include:

- Counting and hence limiting the coverage of some parts of the code to n times during the model checking process.
- The main loop of the program was executed k times in a previously searched execution sequence. We may want to limit the checks to include only executions where the number of iterations are *smaller* than k. (Note that this requires two new variables: a history variable that counts the number of iterations in the current execution, and an auxiliary variable that preserves this value for comparison with other iterations.)

The value of the auxiliary variables in a state is a function of the sequence of states as discovered during the search so far, not necessarily limited to a single execution sequence. Because of their ability to keep values between different execution sequences, auxiliary variables are very useful for achieving various test coverage criteria, e.g., they can be added to record how many times program statements have been traversed. Auxiliary variables may be updated according to history and program variables, but not vice versa.

In order to define history or auxiliary variables, the user can prefix the relevant variable declaration with `history` or `auxiliary`, respectively.

New Search Control Constructs

`commit` Do not backtrack further from this point. This construct can be used when it is highly likely that the current prefix of execution that has accumulated in the search stack will lead to the sought-after counterexample.

`halt` Do not continue the search further beyond the current state. Perform backtrack immediately, and thereafter search in a different direction. This construct is useful for limiting the amount of time and space used.

`report ''text''` Stop the search, type the given text and report the context of the search stack.

`annotate` { **annotation** } Add **annotation**, a piece of code that is responsible to update the history or auxiliary variables. The annotating code may itself include a conditions and a transformation, hence it has the form

`when` (**condition**) **basic_stmt_plus**

Either the condition or transformation is optional. The transformation in **basic_stmt_plus** can include code that can change the history and auxiliary variables or consist of the new constructs `commit`, `halt` and `report`. Since the transformation can include loops, it is the responsibility of the annotator not to introduce nontermination. If the annotation condition holds, the transformation is executed between two atomic actions, according to the location of the `annotate` construct within the checked program. Its execution is *not* counted within the number of atomic steps.

`with` { **stmt** } `annotate` { **annotation** } Similar to the previous statement, except that this annotation is added to every atomic action translated from **stmt**. The condition part of the annotation is conjoined with the condition of the original atomic action. It is possible that there are several nested annotations for the same atomic statement. In this case all the relevant conditions are conjoined together. If this conjunction holds the collection of statements in the transformation parts of all the relevant annotations are executed according to some predefined order.

We can summarize the new construct with the following BNF grammar:

stmt $\longrightarrow$ **annotated** | **basic_stmt**

basic_stmt $\longrightarrow$ `while` (**condition**) { **basic_stmt** } |
`if` (**condition**) { **basic_stmt** } |
{ **list_basic_stmt** } | ...

annotated $\longrightarrow$ `with` { **stmt** } `annotate` { **annotation** } |
`annotate` { **annotation** }

annotation $\longrightarrow$ **basic_stmt_plus** | `when` (**condition**) **basic_stmt_plus** |
`when` (**condition**)

basic_stmt_plus $\longrightarrow$ **basic_stmt** | `commit;` | `halt;` | `report` ‘‘ **text** ‘‘`;`

Accordingly, we allow a nested annotation. On the other hand, the annotating code can include some special commands (`commit`, `halt`, `report`) but cannot itself include an annotated statement. Of course, this is only a suggested syntax. One may think of variants such as allowing to annotate the conditions in `if` and `while` statements, or allowing the new commands to appear embedded within arbitrary C commands inside the annotations. In addition to the above constructs, we added some new features to the C programming language. This

includes allowing concurrent threads and semaphores. These constructs allow us to check or test concurrent programs. We are working on enhancing the language accepted by our tool further in order to support different kinds of concurrency constructs.

Examples

Consider the following examples.

```
with {x=x+y;} annotate {z=z+1;}
```

Ordinarily, we would have translated `x=x+y` into an action with some condition that depends on the program counter value. The translation usually needs to generate a name for the program counter variable (e.g., `pc`) and its values (e.g,. `14`), as these are seldom given in the code. We may obtain the following action, where the condition appears on the left of the arrow '$\Longrightarrow$' and the transformation is given on its right.

```
pc==14 ⟹ x=x+y; pc=15;
```

The annotation is represented as the following wrap action, which is related to the atomic action above.

```
true ⟹ z=z+1;
```

Consider now the code

```
with {while (x<=5) {x=x+1;}} annotate {when (t<23) {t=t+1;}};
```

The code of the while loop may produce the following actions:

```
pc==17 && x>5 ⟹ pc=19;
pc==17 && x<=5 ⟹ pc=18;
pc==18 ⟹ x=x+1; pc=17;
```

The first action checks the loop-exit condition (which is simply the negation of the loop condition). If this holds, the program counter obtains a value that transfers control to the action outside of the loop. The second action checks that the loop condition holds and subsequently transfers control to the first (and only, in this case) action of the loop body. The third action consists of the loop body. Assume that `t` is defined as an auxiliary variable and is initialized to 0. The condition of the wrap action for the annotation is `t<23`, and the transformation increments `t`. This can be used to restrict the above three atomic actions from executing more than 23 times over all the checked executions. If `t` is a history variables, we only allow 23 increments of `t` within any execution. If its an auxiliary variable, we only allow 23 increments over the entire model checking process. The wrap action is therefore:

```
t<23 ⟹ t=t+1;
```

When we execute the above loop actions, `t<23` becomes an additional conjunct, which needs to hold in addition to the atomic action condition. When it does, `t` is incremented, in addition to the effect of the atomic action transformation. Thus, when `t` becomes 23, these actions will become disabled. If this is sequential code, this will disable the search to continue at this point, causing an immediate backtrack. In concurrent code, actions from another process may still be enabled.

The annotation does not apply only to the body of the loop, but also to the actions associated with controlling it. In order to annotate only the increments of `x`, we can use the following:

```
while (x<=5) {with {x=x+1;} annotate {when (t<23) {t=t+1;}}};
```

This will result in exactly the same three actions as above (in general, the atomic actions representing the code are independent of the annotating code). We also have the same wrap action as before, namely

$$\texttt{t<23} \implies \texttt{t=t+1;}$$

Only that this time, this wrap action is only associated with the third action, i.e., the one representing the loop body. Note that if there are other processes, they might not be blocked once `t` reaches 23, and the search can continue. To cause the other processes to be blocked, we can use

```
while (x<=5)
   {with
       {with {x=x+1;} annotate {when (t<23) {t=t+1;}}}
            annotate { when (t≥23) {halt;}}}
```

This will generate, in an addition to the above wrap action, another wrap action, namely

$$\texttt{t}\geq\texttt{23} \implies \texttt{halt;}$$

In this case, the annotation does not block the action when t becomes 23, but rather performs a `halt`, which causes some internal procedure in the search engine to block the search from the current state and induces an immediate backtrack.

3.2 Implementation

We compile a collection of C threads, each interpreted as a concurrent process, into a set of actions. This results in two procedures per each action, one representing the condition and the other representing the transformation. The translation reuses text from the original C code, e.g., conditions and assignments. In this way we do not transfer the code from one language to another, but rather reuse as much as possible of the original code. This is important as modeling and translating are a common source for discrepancy between the checked code and the model. However, using text from the original code is not enough, e.g.,

checks and assignments to program counters (as well as message buffers), which do not appear explicitly in the code, are added.

We need to represent the relation between the procedures representing atomic actions and the procedures representing wrap actions. This is a 'many-to-many' relation. Accordingly, a wrap action can be related to several atomic actions, while an atomic action can be annotated by multiple wrap actions. The program and history variables (but not the auxiliary variables) defined in the checked system are represented in the code obtained by the compilation within a single record (structure) `state`. Thus, if the checked code has a definition

```
int x, y, z;
```

then we have a record `state` with integer fields `x`, `y` and `z`. The search stack consists of records of the same type as `state`. The history variables are kept as part of each state, in the same way as the program variables. In this way, when backtracking, the value of the history variables is rolled back. Auxiliary variables are treated in a different way. They are not part of the structure representing the states. Their value is preserved and updated between different execution sequences despite backtracking.

There are three fixed procedures: `halt`, `commit` and `report`. They are called within some of the wrap actions, when such a construct appears as the corresponding annotation. In this way, once these procedures are called, the search engine takes the appropriate measures to induce immediate backtracking, disallow backtracking from the current state, or stopping the search and reporting the contents of the search stack, respectively.

3.3 Preserving the Viability of Counterexamples

The set of sequences allowed by a specification B, given e.g., by a state machine (automaton) or a formula, is denoted $\mathcal{L}(B)$. The correctness criterion for a system A to satisfy the specification B is

$$\mathcal{L}(A) \subseteq \mathcal{L}(B). \tag{1}$$

It is often simpler and more convenient to provide the complement specification (for a logic based specification, we just have to prefix it with negation) $\overline{B}$, such that $\mathcal{L}(\overline{B}) = \overline{\mathcal{L}(B)}$. We can thus equivalently check for

$$\mathcal{L}(A) \cap \mathcal{L}(\overline{B}) \neq \emptyset. \tag{2}$$

The annotations generates a machine (automaton) A'. We do not include in A' sequences trimmed by the annotations. That is, sequences ending with a `halt` instruction or ending due to the disabledness of all actions caused by the addition of the conditions from wrap actions. Reporting such partial sequences can result in a wrong conclusion about that system. For example, if we want to check whether a state with $x > y$ is eventually reached, the language of $\overline{B}$ consists of sequences in which a state with $x > y$ always holds. It is possible that

due to trimming of a sequence we obtain such a partial sequence, which would otherwise have continued to a state in which $x > y$. Reporting such a sequence as a counterexample would comprise a false negative.

Each execution σ' of A' is related to some execution σ of A in the following way: the states of σ' may include additional variables (history and auxiliary). In addition, it is possible that σ' includes additional actions that did not exist originally in A (due to `annotate` statements without a `with` clause). Let $proj_A(\sigma')$ be the projection of σ' that removes all such variables components from the states, and the additional actions. Note that since the annotations can change only the values of the history and auxiliary variables, removing actions that are related only to the annotation from the projection on the program variables is the same as eliminating some of the 'stuttering' (i.e., the adjacent repetition of the same state) of the execution. Extend now $proj_A$ from sequences to sets of sequences, i.e., let $proj_A(X)$ be $\{proj_A(\sigma')|\sigma' \in X\}$. The code annotation is designed to have the following property:

$$proj_A(\mathcal{L}(A')) \subseteq \mathcal{L}(A) \tag{3}$$

Note that the specification B is compared by the model checker with the projected sequence $proj_A(\mathcal{L}(A'))$. The extra variables and states do not count as part of the execution, but are merely used to control the search. Now, suppose that

$$proj_A(\mathcal{L}(A')) \cap \mathcal{L}(\overline{B}) \neq \emptyset. \tag{4}$$

Then it follows from (3) that (2) holds as well. This means that the annotations do not generate false negatives. They may result in the search being less exhaustive, thus it is not safe to conclude that no erroneous execution exists even if none was found.

4 Experiments and Discussion

The *Sieve of Eratosthenes* is an algorithm for calculating prime numbers. The original program works with message passing. Our implementation works so far with shared variables and hence is simulating message passing using semaphore variables (which were added to the language). In its parallel version, there is a leftmost process that is responsible for generating *integer* numbers, starting from 2, and up to some limit P. There are in general N middle processes (and thus altogether $N+2$ processes), each responsible for keeping one prime number. The ith process is responsible for keeping the ith prime number. Each middle process receives numbers from its left. The first number it receives is a prime number, and it is kept by the receiving process. Subsequent numbers, 'candidates' for prime numbers, are checked against that first number: if dividing a new number arriving from left by the first number gives a remainder of 0, this cannot be a prime number, and hence it is discarded; otherwise the new value is sent to the process on the right. Thus, numbers that are not prime are being sifted-out. The rightmost process simply accepts values from the left. The first number

it receives is kept (this is a prime number), while the other numbers are just being ignored. This allows overflow of numbers, when more prime numbers are generated by the leftmost process than there are processes. The algorithm is presented below with two middle processes.

The annotation we use are based on the following observation: there is concurrency in the system, e.g., processes on the left are checking some new candidates for prime numbers while processes on the right are still working on previous candidates. We can limit the concurrency and control the number S of candidate values that can be propagated at the same time in the system. We do this by introducing a new history variable `checked`. The value of `checked` is updated using annotations each time that a candidate value is either sifted out or reaches its final destination in a process. Generating a new candidate on the left is controlled by annotation, allowing the new candidates to be at most S values ahead of the currently checked value. We keep S as a constant with which we control the amount of concurrency we allow in the validation search.

It can be shown that for some class of properties (specifically, those that preserve a partial order equivalence between executions, see, e.g., [13]), there is no loss of information by checking only the annotated version. A proof of a similar program is discussed in [13]. However, such proofs are not always available, and thus we cannot always count on the fact that we will not lose some of the exhaustiveness of the verification.

The experimental results are summarized in the table in Figure 1 for $N = 4$ and $P = 12$. Observe that the biggest reduction in the state space is when $S = 0$, i.e., only one candidate is allowed to progress through the system. Note that repeating the same experiment without the annotation produces the same number of states and transitions as in the last row in the table. The reason for that is that we used the *else* keyword in order to enforce an annotation to be executed when a condition *does not hold*. This *else* was not necessary in the original program, and its introduction added some more states. For sanity check, we have also run an unannotated version of the program with an empty *else* statement. Indeed in this case the number of states in this case coincides with the worst case of annotation. We list below the sieve program with $N = 2$ (note that the experiments were done with $N = 4$).

S Values	States	Transitions
no annotation:	123403	510062
no annotation with *else*:	139267	578782
0	11567	45233
1	51567	209707
2	105983	439121
3	134443	559080
4	139123	578218
5	139267	578282

Fig. 1. The experimental results

```
main () {
   int q0, q1, q2;
   semaphore w0 = 1, r0 = 0;
   semaphore w1 = 1, r1 = 0;
   semaphore w2 = 1, r2 = 0;
   history int checked = 1;
   thread left {
      int counter = 2;
      while(counter <= P) {
         P(w0);
         q0 = counter;
         V(r0);
         with { counter = counter + 1; }
              annotate { when ((counter-checked) <= S); }
      }
   }
   thread middle1 {
      int myval1, nextval1;
      P(r0);
      with { myval1 = q0; } annotate { checked = checked+1; }
      V(w0);
      while(1) {
         P(r0);
         nextval1 = q0;
         V(w0);

         if((nextval1 % myval1) != 0) {
            P(w1);
            q1 = nextval1;
            V(r1);
         }
         else
            annotate { checked = checked+1; }
      }
   }
   thread middle2 {
      int myval2, nextval2;
      P(r1);
      with { myval2 = q1; } annotate { checked = checked+1; }
      V(w1);
      while(1) {
         P(r1);
         nextval2 = q1;
         V(w1);
         if((nextval2 % myval2) != 0) {
```

```
            P(w2);
            q2 = nextval2;
            V(r2);
         }
         else
            annotate { checked = checked+1; }
      }
   }
   thread right {
      int next;
      while(1) {
         P(r2);
         with { next = q2; } annotate { checked = checked+1; }
         V(w2);
      }
   }

}
```

It is interesting to observe that the SPIN system obtains such a reduction automatically, by applying a built in partial order reduction algorithm [12]. By applying our insight and using the annotation, we obtained a similar reduction without implementing the partial order algorithm. We do not suggest the use of annotations to replace automatic and effective state space reduction. We are merely demonstrating the potential and flexibility of our annotation mechanism. An important observation is that by using a parameter (denoted by S in this example), we controlled the exhaustiveness of the search. It can be proved that in this specific example, the executions searched are anyway equivalent to the ones that were explored. In other cases we can lose exhaustiveness.

We presented a methodology and a syntax for annotating verified programs. The program annotations can be used to direct the search and suppress part of the state space. It can be used to compromize the exhaustiveness of model checking, when the state space is too large to complete.

It is important to note that the annotations, when not used carefully, may also increase the size of the state space. For example, consider the case that we use a history variable to keep the number of messages that have arrived. This encoding can result in multiple states having the same value for the program variables, but different values for the history variable. There are cases where we may allow introducing history variables that would cause some state repetition, while on the other hand gaining a lot of reduction.

Future work is planned based on the fact that annotating the code for automatic verification can be used in many ways. One can use the annotations to extend the capabilities of a simple model checker that searches for deadlocks into one that checks safety properties, given as a deterministic finite automata. In this case, the automaton behavior is encoded within an annotation that is applied to the entire code. Some further constructs are conceivable. For exam-

ple, we may allow checking whether the current state is already present in the search stack, or permit a nondeterministic choice between annotation actions. With such constructs, we can replace some fixed model checking algorithms, e.g., checking for temporal logic properties and various kinds of reductions, within annotations that are automatically generated. This can allow us an open ended model checker, with flexible capabilities.

References

1. K. R. Apt, E. R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1991.
2. M. Ben Ari, Principles of Concurrent and Distributed Programming, Prentice Hall, 1990.
3. E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking*, MIT Press, 2000.
4. E. M. Clarke, E. A. Emerson, S. Jha, A. P. Sistla, Symmetry reductions in model checking, Computer Aided Verification 1988, LNCS 1427, Vancouver, BC, 147–158.
5. E.W. Dijkstra, Guarded Commands, Nondeterminacy and Formal Derivation of Programs, *Communications of the ACM*, 18(1975), 453-457.
6. C. Eisner, D. Peled, Comparing Symbolic and Explicit Model Checking of a Software Systems, SPIN 2002, Grenoble, France, LNCS 2318, Springer Verlag, 230–239.
7. D. Evans, J. Guttag, J. Horning, Y. M. Tan, LCLint: A tool for using specifications to check code, Proceedings of the SIGSOFT symposium on the foundations of software engineering, 1994, 87–96.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata, Extended Static Checking for Java, PLDI 2002, 234–245.
9. N. Francez, *Program Verification*, Addison-Wesley, 1992.
10. N. Francez, *Fairness.* Springer-Verlag, 1986.
11. G. Holzmann, The Spin Model Checker, Primer and Reference Manual, Addison-Wesley, to appear, 2003.
12. G. J. Holzmann, D. Peled, An improvement inFormal Verification, FORTE 1994, 197–211.
13. S. Katz, D. Peled, Defining conditional independence using collapses, Theoretical Computer Science 101(1992) 337–359.
14. R. Kurshan, Computer-Aided Verification of Coordinating Processes, Princeton University Press, 1995.
15. R. Lazic, D. Nowak, A unifying approach to data-independence, Conference on Concurrency Theory (CONCUR) 2002, LNCS 1877, 581–595.
16. S. Edelkamp, S. Leue, A. Lluch-Lafuente, Directed explicit-state model checking in the validation of communication protocols, Software Tools for Technology Transfer, to appear.
17. Z. Manna, A. Pnueli, How to cook a temporal proof system for your pet language, Principles of Programming Languages 1983, Austin, Texas, 167-176.
18. K. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.
19. G. J. Myers, *The Art of Software Testing*, Wiley, 1979.
20. S. Owicki, D. Gries, Verifying properties of parallel programs: an axiomatic approach, CACM 19(1976), 279–285.
21. D. A. Peled, *Software Reliability Methods*, Springer Verlag, 2001.
22. A. Pnueli, The temporal logic of programs, 18th IEEE symposium on Foundation of Computer Science, 1977, 46–57.

23. P. Wolper, Expressing interesting properties of programs in propositional temporal logic, Principles of Programming Languages 1986, St. Petersburg Beach, Fl, 184–193.
24. M. Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, Proceedings of the 1st Annual Symposium on Logic in Computer Science IEEE, 1986, 332–344.