THE EFFECT OF UNCONTROLLED CONCURRENCY ON MODEL CHECKING

Donna M. Carter, Ramazan Aygun, Glenn Cox, Mary Ellen Weisskopf, Letha Etzkorn

Donna Carter
COLSA Corporation
Email: dmcarter@colsa.com

Dr. Ramazan Aygun (Corresponding Author)
Computer Science Department
Technology Hall, N360
University of Alabama in Huntsville
Huntsville, AL 35899
Email: raygun@cs.uah.edu
Phone: 1 (256) 8246455
Fax: 1 (256) 8246239

Dr. Glenn Cox
Computer Science Department
University of Alabama in Huntsville
Huntsville, AL 35899
Email: gcox@cs.uah.edu

Dr. Mary Ellen Weisskopf
Computer Science Department
University of Alabama in Huntsville
Huntsville, AL 35899
Email: weisskop@cs.uah.edu

Dr. Letha Etzkorn
Computer Science Department
University of Alabama in Huntsville
Huntsville, AL 35899
Email: letzkorn@cs.uah.edu

KEYWORDS

Model checking, multimedia synchronization, uncontrolled concurrency, PROMELA/SPIN

ABSTRACT

Correctness of concurrent software is usually checked by techniques such as peer code reviews or code walkthroughs and testing. These techniques, however, are subject to human error, and thus, do not achieve in-depth verification of correctness. Model checking techniques, which can systematically identify and verify every state that a system can enter, are a powerful alternative method for verifying concurrent systems. However, the usefulness of model checking is limited because the number of states for concurrent models grows exponentially with the number of processes in the system. This is often referred to as the "state explosion problem." Some processes are a central part of the software operation and must be included in the model. However, we have found that some exponential complexity results due to uncontrolled concurrency introduced by the programmer rather than the intrinsic characteristics of the software being modeled. We have performed tests on multimedia synchronization to show the effect of abstraction as well as uncontrolled concurrency using the Promela/SPIN model checker. We begin with a sequential model not expected to have exponential complexity but that results in exponential complexity. In this paper, we provide alternative designs and explain how uncontrolled concurrency can be removed from the code.

1. INTRODUCTION

The verification of concurrent systems is important since failures in software can have fatal and costly results. In particular, asynchronous, multithreaded, and distributed systems require synchronization and reliable communication protocols and in these systems, the interaction among software/hardware modules should be coordinated to avoid unexpected failures. The verification of software and concurrent systems has been widely studied in [1] and [2]. Also, verification methods used in different parts of the world have been surveyed [3], [4], and [5]. Several verification tools have been developed for verification of systems implemented in popular languages like Java [6], [7], and [8]. Some of the verification methods that are employed are code inspections/walkthroughs, pair programming, automated static analysis tools, coverage, capture/playback, model checking, and development testing tools [1]. We are interested in model checking verification since model checking is more rigorous than most other verification techniques because it can check all possible states a model can enter. There are usually three steps in model checking verification: modeling, programming in the language of a model checker in order to implement a given model, and verification. Since all three steps are likely to be handled by a single person, we refer to a programmer as a person who models, implements, and verifies the model using a model checker. In this sense, a programmer is able to apply abstraction to the model as well as verify the model.

1.1 State Explosion

Exponential growth in the number of states that must be checked, usually referred to as State Explosion, is mentioned as one of the major bottlenecks when applying model checking to software [9]. Because of state explosion, the size of a finite state model must be small [10]. In [9], two approaches are proposed to deal with state explosion: abstraction and slicing. Both slicing and abstraction are performed based on an input property. In a “slice,” the parts of a program that do not affect the observable features are removed. For data abstraction, the properties of variables are minimized. If the reachability of a state from any other state needs to be checked, this type of slicing and abstraction may not be useful.

State explosion may result from the inherent complexity of a model as well as from unsophisticated coding by the programmer. State explosion often occurs in complex models.. Unfortunately, using current techniques, it is not usually possible to state definitively whether there is going to be state explosion or not until the program model is completed and the model checker is run on the model. If there is state explosion, it is not clear whether or not to continue the model checking process. To the best of our knowledge, although model checkers like Simple Promela Interpreter (SPIN) [11] use on-the-fly state generation and partial order reduction to deal with state explosion, no research on how to deal with state explosion from the programmer's perspective has been performed. That is, stylistic techniques that the model developer can use to reduce the magnitude of the state explosion problem have not been heretofore developed.

1.2 Uncontrolled Concurrency

In this paper, we examine components of model complexity that are introduced by the programmer's stylistic choices and we suggest how to reduce this additional complexity. If the programmer does not remove uncontrolled concurrency, model checkers are likely to face state explosion.

Uncontrolled concurrency within a model is unintended concurrency by the programmer. In other words, if the programmer had been aware of the uncontrolled concurrency, the programmer would have attempted to remove it. If a model has uncontrolled concurrency, the model checker may face state explosion during verification. Even though a model is correct – meaning all properties can be satisfied -- there can still be uncontrolled concurrency. If a model is correct but has uncontrolled concurrency, a programmer will unnecessarily simplify the model by applying abstraction to perform verification on a larger scale. We show that careless code creation resulting in uncontrolled concurrency can lead to exponential complexity. Reducing the complexity on one part of the system enables the programmer to perform more thorough tests on other parts of the system. Note that we do not reduce the complexity of the original model. In this paper, we try rather to identify the existence of uncontrolled concurrency and then remove it. Therefore, we point out that the uncontrolled concurrency needs to be removed before applying any simplification to the model.

1.3 PROMELA/SPIN

This paper examines the Simple Promela Interpreter (SPIN) model checker and Process Meta-Language (Promela) in identifying the complexity of verification. We have chosen Promela/SPIN as the model checker since SPIN is expected to work well on asynchronous software systems, multi-threaded software, and distributed algorithms [11]. The SPIN model checker is a tool for verifying distributed software models. It was developed at the Bell Labs Computing Sciences Research Center in 1980 and has been available as open source since 1991 [13]. After its significant contributions to model checking, SPIN was awarded the ACM System Software Award in 2001. SPIN thoroughly checks whether a model contains the essential elements of a distributed system design [11]. Promela is the specification language used by SPIN [11]. Promela is not a programming language but a language for building formal verification models. A model is an abstraction and contains correctness properties and other elements that are not part of a program's implementation [11]. The features found in Promela aid in building a high-level distributed system model [11]. Programs implemented in traditional programming languages such as in C can be mapped to PROMELA, and the SPIN model checker can be applied for verification [14].

This study examines the number of states stored, memory usage and time. A state contains a program's description such as variable values, process counters, etc. [15]. We provide examples on how to deal with state explosion from the programmer's perspective. We perform tests on multimedia synchronization whose purpose is to provide correct in-order delivery of streams from a server to a receiver. Multimedia synchronization using model checking has been previously studied in [12]. Here, we examine the number of processes, or proctypes in Promela/SPIN, as well as the number of states per proctype. The number of states per proctype corresponds to the abstraction of the model. For multimedia synchronization, we focus on a sequential model that is not expected to have exponential complexity. SPIN keeps track of possible model states using state space methods [11]. State space methods construct all states and state transitions within a system that can lead to state explosion. Most systems will have a large number of states that can grow exponentially [16].

This paper is organized as follows: Section 2 provides brief information about the Promela keywords used in this paper; Section 3 provides different design strategies for multimedia presentations with a focus on sequential presentations; Section 4 compares different designs and discusses the reasons for exponential complexity; Section 5 discusses the experiments and the last section concludes our paper.

2. BRIEF BACKGROUND on PROMELA

Proctype is a Promela keyword and is used to declare new process behavior [11]. Promela proctypes are either static or active. The keyword “active,” when placed in front of a proctype declaration declares and instantiates a process [11]. Static proctypes need to be called from a function.

The syntax of a proctype is

proctype *name* ([*decl_lst*]) { *sequence* }[11]

where *decl_lst* is the declaration list and *sequence* is a block of code enclosed within curly braces [11].

In Promela, there is a reserved proctype called *init*. This is similar to the *main* function in C/C++ since the *init* proctype is the default starting proctype and does not need to be initialized. All other proctypes can be initialized from the *init* proctype if they are not active.

A do loop is a repetitive construct [11]. The do loop syntax is:

do :: *sequence* [:: *sequence*] * od

where there can be one or more sequences written in the do loop [11]. A double colon precedes each separate option sequence [11].

A Promela do loop begins with the keyword *do* and ends with the keyword *od*. Each do loop has a condition statement that evaluates to true (1) or false (0) [11]. Promela's do loop operates differently than in a programming language. With Promela, do loop options execute randomly, not sequentially. Only one option within a do loop is selected for execution each time the do loop is executed. After an option is executed, the do loop continues execution from the beginning of the do loop. The do loop is exited only by using a break statement.

The double colon in front of the condition statements in a do loop makes them guard statements [11]. An arrow symbol (->) following a guard statement is equivalent to a semicolon in Promela. The code following the arrow symbol is not executed until the guard statement evaluates to true. If the guard statements do not evaluate to true, code execution returns to the beginning of the do loop.

3. MULTIMEDIA SYNCHRONIZATION in PROMELA/SPIN

Two types of multimedia synchronization are usually analyzed: fine-grained synchronization and coarse-grained synchronization. Fine-grained-synchronization refers to tight synchronization between different streams. Lip synchronization between video and audio is an example of fine-grained synchronization. In this paper, we are interested in coarse-grained synchronization. Coarse-grained synchronization synchronizes the points at which streams start and end. Synchronized Multimedia Integration Language (SMIL) [17] is an XML-based language that enables the specification of coarse-grained synchronization. During a multimedia presentation, there could be streams that are played in parallel as well as sequentially. By an event mechanism, the streams may be notified when to start.

In this section, we examine a model that would not be expected to have exponential complexity, yet yields exponential complexity. The sequential presentation of multimedia streams has linear complexity since each stream starts after the previous stream ends. A stream should notify the next stream to start in the sequence. The notification in our models is handled using streams' state variables. In other words, each stream is waiting for a condition to be satisfied.

We define three general designs to express sequential multimedia presentations in Promela: embedded, wake-up, and chain. The embedded design has only one proctype, and all the components of the streams are embedded in a single proctype. In the wake-up design, there is one proctype per stream, and each proctype is initiated at the same time. However, each proctype is idle until the previous proctype in the sequential presentation finishes its execution. In the chain design, there is again one proctype per stream and each proctype is initiated by the previous proctype in the sequential presentation.

Figure 1 shows the sequential execution of three streams with guard conditions as a state graph. Stream p_1 is the first process to be executed. As long as the endstate of p_1 is not reached, p_1 is executed. When p_1 reaches its endstate, the guard condition for p_2 ($stateA_1$==endstate) is satisfied and p_2 is allowed to start. In the same way, p_3 is not allowed to start until p_2 reaches the endstate.

3.1 Wake-up Sequential Presentation Design

Figure 2 shows the Promela code for the wake-up sequential presentation design. Each proctype p_i corresponds to a stream and contains one do loop that increments its state through a system variable $stateA_{,i}$ where i represents the i^{th} stream, $1 \leq i \leq n$, and n is the number of streams. The state variable for each stream is initialized to 0 in the Promela programs. The variable *endstate* determines the maximum number of states a stream can enter, i.e., the maximum value for the number of states of a stream. These states may correspond to frames displayed in a video.

With the conditional statements in place, i.e., (stateA_i == endstate), proctype p_1 executes first while the rest of the proctypes p_i ($i>1$) remain idle until proctype p_1 completes executing. Proctype p_1 completes executing when variables stateA_1 and endstate are of equal value. Proctype p_2 remains idle until the control structure in p_2, (stateA_1 == endstate) evaluates to true. Accordingly, proctype p_3 runs after proctype p_2 completes executing, i.e., when (stateA_2 == endstate) (Figure 2).

There are two options within each do loop. The first is a guard statement where the state variable is checked to see if it is less than the endstate variable. The second option is an else statement. The else statement is also a conditional statement. The sequence following the else statement is executed if the else statement is the only statement that is executable in the do loop [11]. The break statement causes the code execution to go to the end of the do loop [11].

3.2 Embedded Sequential Presentation Design

The embedded sequential presentation design includes all of the variables used in the wake-up sequential presentation design except that the do loops are embedded within one proctype, p_1. Initialization remains the same. Each of the do loops sequentially increments the corresponding state variable. Sample Promela code for three streams is shown in Figure 3.

3.3 Chain Sequential Presentation Design

The chain sequential presentation design is very similar to the wake-up design. In this case, proctype p_i is initiated in proctype, $p_{(i-1)}$ (where $2 \le i \le n$) prior to the completion of $p_{(i-1)}$. The guard statements waiting for the completion of the previous proctype are not needed anymore since the previous proctype only initiates just before its completion. Figure 4 shows sample

Promela code for the chain design. (Notice the *run* statements at the end of each proctype). The init function only needs to call proctype p_1. Sequential functionality remains the same.

4. UNCONTROLLED CONCURRENCY AND COMPARISON OF DESIGNS

In this section, we compare the wake-up, embedded, and chain designs with respect to similarities in abstraction and concurrency. Then, we explain uncontrolled concurrency and briefly explain how to remove it.

4.1 Comparison of Designs for Multimedia Synchronization

The wake-up design is a more realistic abstraction than the chain and embedded designs since the streams are waiting for an event to start. In this abstraction, an event is handled through state variables and guard statements. The embedded design provides a higher-level abstraction by embedding all streams into a single proctype. This drastically reduces the complexity. However, one may think that embedded abstraction is not realistic since all streams are expressed by exactly one proctype. The third alternative, the chain design, provides one proctype per stream; however, it does not initiate a proctype until the previous proctype reaches its conclusion. The chain design lies somewhere between the embedded and wake-up designs in terms of abstraction level but is closer to the wake-up design.

In the embedded design, it is clear that there is no interaction among the streams at all. Therefore, the programmer assumes that the interactions among streams are (or will be) handled properly and probably focuses on other components of the system verification. It is possible to see a kind of interaction among streams in the wake-up design using guard statements. For the chain design, it is possible to check whether a proctype is starting another proctype.

4.2 Uncontrolled Concurrency and Its Removal

When a programmer creates a model, the programmer has a general, coarse idea about the number of states that could be generated. In some cases, the programmer is certain that the model is correct but may use a model checker just to clarify that nothing unexpected happens. The model checker may determine critical states that are never estimated. Because of an unsophisticated model representation, there are unnecessary states to be checked that greatly increase state explosion. By “unnecessary” we mean the programmer is not interested in these states and, if the programmer had been aware of them, would try to avoid them. These unnecessary states cause uncontrolled concurrency.

We have run all designs in SPIN’s interactive mode to study how each design is executed by SPIN. In interactive mode, SPIN executes the code and whenever it reaches a decision point, it asks the programmer which path (or choice) to take. For the embedded and chain designs, SPIN never asked the programmer to make a choice since there was exactly one way to execute at all times. However, for the wake-up design, SPIN has asked the programmer to make choices. This shows that although the presentation is sequential, SPIN found parallel executions and, therefore, more than one possible execution path. In the wake-up design, we realized that the proctype p_i is ready to start just after stateA$_{(i-1)}$ of p$_{(i-1)}$ is incremented and becomes equal to endstate. There are three more steps for proctype p$_{(S-1)}$ to terminate: else, break, and termination. These three extra steps can be executed while future proctypes are being processed. It is possible that proctype p$_1$ is the last proctype to terminate its execution.

The wake-up design is the most realistic of the three designs. Can the wake-up design be improved? To improve it, the uncontrolled concurrency caused by the else statement should be removed. The only way to correct this is to ensure that the state of the proctype becomes equal to endstate in the last statement of the proctype. We update the proctype as follows (the code for p$_1$ is given as an example):

```
proctype P1(){
    do
      :: ( stateA1 < (endstate-1) ) ->
                  stateA1++;
      :: else -> break;
    od;
    stateA1++;
}
```

In this example, the loop termination condition is $stateA_1$==(endstate-1) rather than $stateA_1$ == endstate. When the loop's execution is completed, the next proctype cannot start since stateA1 is equal to endstate-1. The last statement in the proctype assigns the state variable to endstate and the next proctype becomes eligible to take steps. We call this design "wake-up design with controlled concurrency (wake-up CC)". We have tested the wake-up CC design in interactive mode and SPIN never asks the programmer for a path choice.

Note that none of the designs violate any property to be verified. Normally, in this type of verification, the programmer is only interested in whether a stream starts after the end of the previous stream. In all of the designs, the sequential presentations are obtained as expected. This shows that all of the designs are indeed correct.

5. EXPERIMENTS

Sample execution of the embedded and wake-up designs is given in the appendix. The examples chosen for our experiments are relatively simple. We chose them so the issues related to uncontrolled concurrency can be pointed out clearly.

5.1 The Effect of Abstraction on Streams

The state variable for each stream maintains the number of states a stream may enter. For example, for a video stream, a state may correspond to the display of a frame (picture). Since there are 30 frames per second in a typical video, there are 10,800 frames in one hour of video. We may apply abstraction in such a way that when the state variable is increased, one second of video is displayed. The abstraction actually indicates the number of frames corresponding to each state variable unit in our example.

To realize the effect of abstraction on the number of states per stream, we have kept the number of streams constant. In this set of experiments, the number of streams is set to 3. The endstate is initialized to a value of 100 and is incremented by 100 in each consecutive program until endstate is equal to 1,000. Endstate was incremented by 100 with each successive file in order to obtain the one-minute runtime goal.

The number of states that are generated by SPIN. Figure 5 displays linear growth for the number of states stored for the wake-up, chain, embedded, and wake-up CC presentations for three streams. In the embedded design, when the endstate is 100, there are 1,423 states stored. When the endstate is 1,000, there are 14,023 states stored. In the wake-up design, when endstate is 100, there are 609 states stored in the one-proctype model whereas, when the endstate is 1,000, there are 6,009 states stored. In the chain design, when endstate is 100, there are 611 states stored whereas, when endstate is 1,000, there are 6,011 states stored. In the wake-up CC design, when endstate is 100, there are 608 states stored in the one-proctype model whereas, when endstate is 1,000, there are 6,008 states stored.

Memory Usage by SPIN. Figure 6 displays the memory usage for the chain, embedded, wake-up, and wake-up CC designs. As the value of endstate increases, memory usage appears to increase linearly. Memory usage for the wake-up design is 2.622 MB when the endstate is 100 and increases to 3.134 MB when the endstate is 1,000. For the embedded design, the memory usage is 2.622 MB when the endstate is 100 and 2.827 MB when the endstate is 1,000. For the chain and wake-up CC designs, the memory usage is almost the same as in the embedded design.

Time. Figure 7 displays the real time used for the embedded, chain, wake-up and wake-up CC designs. Real time used appears to increase linearly with the endstate.

5.2 The Effect of Number of Streams

The number of streams corresponds to the number of proctypes in the wake-up, wake-up CC, and chain designs, and the number of do loops in the embedded design. For the embedded sequential design, when the number of streams in the program is 50, SPIN outputs the message that one of 400 states is unreached. If some states have not been reached, the full search of the state space has not been completed. To have a fair comparison, we have to make sure that all executions do not have any states that have not been reached. To overcome the “unreached state” problem, some adjustments are needed either in the program or when compiling and executing SPIN output. Making these adjustments would make the comparisons unfair. When 49 proctypes are used, the runtime has only reached .032 seconds. Since the experiment could not continue without receiving the unreached state message, the SPIN results for streams 4 through 49 have been graphed.

For the chain and wake-up CC designs, the programs are tested until there are 49 proctypes in the file. At 50 proctypes, the “max search depth too small” error is displayed and there is one unreached proctype. Again, to have fair comparisons, the results for the 4-proctype through 49-proctype programs are graphed. For the wake-up design, the runtime reaches the one-minute mark when the number of streams is 15. Since the running time increases, we have not done any further experiments for the wake-up design for more than 15 streams.

States. Figure 8 displays the number of states stored for the wake-up design. When there are four proctypes in the program, 3,047 states are stored. The last program in the experiment contains 15 proctypes and results in 6,651,700 states stored.

Figure 9 displays linear growth of the number of states stored for the embedded, chain, and wake-up CC designs. When the program contains four streams, there are 811 states stored for the embedded design. The last program in this experiment contains 49 do loops for the embedded design and results in 9,901 states stored. When there are four proctypes in the program, 814 states are stored for the chain design. The 49-proctype program results in 9,949 states stored. When there are four proctypes in the program, 810 states are stored for the chain and wake-up CC designs. The 49-proctype program results in 9,900 states stored in both the chain and wake-up CC designs. The number of states stored for the chain, embedded, and wake-up CC designs are very close to each other.

Memory Usage. Figure 10 displays the memory usage for the wake-up design. The memory usage appears to remain relatively constant until the program instantiates eight proctypes, after which the memory usage clearly grows exponentially until 747.582 MB are used in the 15-proctype program. Figure 11 displays the memory usage for the embedded, chain, and wake-up CC designs. For the embedded, chain, and wake-up CC designs, the memory usage pattern contains similar periods of relatively constant memory usage with small increases in memory usage. Memory usage is 2.622 MB for the four-proctype program and increases to 5.284 MB for the 49-proctype program.

Time. Figure 12 displays real time for the wake-up design. Real time appears to remain relatively constant until nine proctypes have been instantiated. Exponential growth in the runtime continues for the remainder of the programs until 118.032 seconds is reached with the 15-proctypes program. Figure 13 displays the real time used for the embedded, chain, and wake-up CC designs. Real time used appears to increase at regular intervals. Real time is .005 seconds with four proctypes and increases to around .070 seconds with the 49-proctypes program.

5.3 Analysis of Complexity from Experiments

Table 1 compares the data results for varying endstates for the embedded, wake-up, wake-up CC and chain designs. It can be seen from the table that the wake-up model contains more states stored, total actual memory used and longer runtimes than the embedded, chain, and wake-up CC designs.

Table 2 shows the results for the wake-up design for different numbers of streams (processes). The wake-up design displays exponential growth in complexity. The number of states stored for the 15-proctype program is nearly 2,184 times the number of states stored for the four-proctype program. Memory usage escalates from 2.724 Mb in the four-proctype program to 747.582 Mb in the 15-proctype program. Runtime increases from .011 seconds in the four-proctype program to 118.032 seconds in the 15-proctype program.

Table 3 displays the sample outputs for the embedded and chain designs for different number of stream (processes). The results for the embedded and chain designs are very similar to each other. The growth from four to 49 do loops in the proctype remains linear.

From the given results, it is clear that the runtime complexity of the embedded design is O(n) where n is the number of streams. It can also be noted that the runtime complexity for the chain and wake-up designs is also almost O(n). It is also clear that the runtime complexity of the wake-up design is not linear. The complexity of the runtime slope for the wake-up design can be expressed as $C(an^2 + bn + c) * 2^n$ where C is approximately 1/500, a is 1/24, b is -1/4 and c is 4/3. Since the runtime slope increases exponentially, the runtime has exponential complexity. It can also be noted that the complexity of number of states and memory usage is also exponential for the wake-up model whereas it is linear for the embedded, chain, and wake-up CC designs.

5.4 Discussion

The experiments on multimedia synchronization are interesting since four different designs are provided and tested for multimedia synchronization. From the programmer's perspective, it is clear that the sequential presentation has linear complexity since the states (or play) of streams do not interact with each other. However, for the wake-up design, the complexity is exponential and state explosion is experienced. In such cases, the programmer is likely to simplify the model rather than remove the uncontrolled concurrency. As mentioned, in the original wake-up design, there are some leftover states for the proctype after notifying the next stream. Because of these leftover states, the design behaves as if it is a parallel execution of streams and causes exponential complexity. When the uncontrolled redundancy is removed, the model can be verified as having linear complexity.

Although increasing the number of proctypes yields exponential complexity, increasing the number of states (endstate) per proctype results in linear complexity. This means that there is another part of the code that introduces the exponential complexity and indicates the existence of uncontrolled concurrency.

Linear complexity can also be achieved in the embedded and chain designs by applying further abstraction. Does the programmer need to apply higher abstraction than before when state explosion is likely to happen? The wake-up CC design is almost the same as the wake-up design; however, the uncontrolled concurrency is removed. The removal of the uncontrolled concurrency reduces the complexity from exponential to linear complexity. The complexity of the wake-up CC design is very similar to the embedded and chain designs in terms of the number of states, memory usage, and runtime.

6. CONCLUSION

In this paper, we examined the complexity of the verification of model checkers by examining sequential multimedia synchronization using one of the model checkers, Promela/SPIN. The use of proctypes with uncontrolled concurrency is crucial as it affects model complexity exponentially. The wake-up, chain, and embedded designs yield the same sequential and equivalent models. Each proctype in each design, except the chain design, waits for a state variable to equal a variable value before executing. However, the wake-up design causes greater complexity.

We have shown that unsophisticated specification with uncontrolled concurrency of sequential streams causes exponential complexity. Uncontrolled concurrency causes additional complexity to the verification of concurrent processes. If the chain and embedded designs are used, the program can contain more proctypes to handle other functionality to minimize program complexity as much as possible. We have further shown that even the complexity of the wake-up design can be reduced significantly if uncontrolled concurrency is removed from the code as in the wake-up CC design.

At this point, we do not have any design guidelines for model checking. In general, to avoid state explosion, the programmer need some design guidelines. Before reducing the number of processes or states, programmer should focus on removing uncontrolled concurrency. Removal of uncontrolled concurrency may help the model checker to verify the specification without facing state explosion. In this paper, we have shown four types of design for sequential presentations. However, in the future, more design strategies need to be developed that could be applicable in various environments. The programmer should consider possible uncontrolled concurrency that can be avoided. We have also shown that existence of uncontrolled concurrency is not obvious. Since most programmers have backgrounds in traditional programming languages, it is hard for to see a programmer that a good programming style in a traditional programming language may actually cause uncontrolled concurrency for model checking. We recommend the following rules of thumb that need to be applied in the given order to deal with state explosion:

1. Apply verification as early as possible to observe the number of resulting states since it is easier to remove redundancy when the size of the code is small.

2. Check for uncontrolled concurrency (if possible, test several processes (proctypes) individually to see if they have any uncontrolled concurrency rather than all proctypes at the same time),
3. Simplify the model
    a. If a model is still complex, apply abstraction on the attributes by reducing the possible set of values a state variable may take, and/or
    b. Reduce the number of concurrent processes.

The relatively simple example described in this paper resulted in extensive uncontrolled concurrency. The application of model checking on more complex multimedia presentations can be found in our early work [12]. We plan to extend our work by analyzing segments of the code that lead to uncontrolled concurrency. This paper did not address the requirements of demanding concurrent models. Rather, we concentrated on uncontrolled concurrency in relatively simple models. Extending this to demanding concurrent models would be an exciting but challenging research topic.

7. REFERENCES

[1] Wojcicki, M. A. and Strooper, P. 2006. A state-of-practice questionnaire on verification and validation for concurrent programs. In *Proceeding of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging* (Portland, Maine, USA, July 17 - 17, 2006). PADTAD '06. ACM Press, New York, NY, 1-10.

[2] Andersson, C. and Runeson, P., Verification and Validation in Industry - A Qualitative Survey on the State of Practice. In Proceedings of the 2002 International Symposium on Empirical Software Engineering, (2002), 37--47.

[3] Geras, A.M., Smith, M.R. and Miller, J. A survey of software testing practices in Alberta. Canadian Journal of Electrical and Computer Engineering, 29, 3 (2004), 183--191.

[4] Groves, L., Nickson, R., Reeve, G., Reeves, S. and Utting, M., A Survey of Software Development Practices in the New Zealand Software Industry. In Proceedings of the 2000 Australian Software Engineering Conference (ASWEC'00), (2000), 189--201.

[5] Ng, S.P., Murnan, T., Reed, K., Grant, D. and Chen, T.Y., A Preliminary Survey on Software Testing Practices in Australia. In Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04), (2004), 116—125

[6] Eytani, Y., Havelund, K., Stoller, S.D. and Ur, S. Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs. To appear in Concurrency and Computation: Practice and Experience (2006)

[7] Havelund, K. and Pressburger, T. Model Checking Java Programs using Java PathFinder. International Journal of Software Tools for Technology Transfer (STTT), 2, 4 (2000), 366--381.

[8] Long, B., Strooper, P. and Wildman, L. A Method for Verifying Concurrent Java Components. To appear in Concurrency and Computation: Practice and Experience (2006).

[9] Hatcliff, J. and Dwyer, M. Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software. June, 2001. Proceedings of CONCUR 2001

[10] Lam, D. N. and Barber, K. S. Comprehending agent software. In *Proceedings of the Fourth international Joint Conference on Autonomous Agents and Multiagent Systems* (The Netherlands, July 25 - 29, 2005). AAMAS '05. 2005.

[11] Holzmann, G. 2004. *The SPIN Model Checker Primer and Reference Manual*. Boston: Addison-Wesley.

[12] Aygun, R., Zhang A. "SynchRuler: A Rule-Based Flexible Synchronization Model with Model Checking", IEEE Transactions on Knowledge and Data Engineering, Volume 17 , Issue 12 (December 2005) Pages: 1706 - 1720

[13] *Department of Telematics*. 2005-2006. [online]. [Accessed July 2006]. Available from World Wide Web <http://www.item.ntnu.no/labs_promela.php>

[14] Wang W., Hidvegi, Z., Bailey Jr, and A., Whinston, A. 2000. E-Process Design and Assurance Using Model Checking. IEEE Computer, Volume 33, No 10, (October 2000) Pages: 48-53

[15] Barland, S., Vardi, M., Greiner, J. 2006. *Modeling Concurrent Processes*. [online]. [Accessed August 2006]. Available from World Wide Web (http://cnx.org/content/m12316/latest/)

[16] Kot, M. 2003. *The State Explosion Problem*. Unpublished.

[17] SMIL. The Synchronized Multimedia Integration Language. http://www.w3.org/AudioVideo/

APPENDIX

Table 4 displays sample sequential executions of the wake-up and embedded models. The execution shown is for the four-proctype and the one-proctype (six do loop) programs. This is an output of SPIN for the corresponding Promela code.

Table 1 Summary of sequential presentation complexity with varying endstate

endstate	# of States Stored				Total Actual Memory Used (Mb)				Real Time (sec)			
	Wake-up	Chain	Embedded	Wake-up CC	Wake-up	Chain	Embedded	Wake-up CC	Wake-up	Chain	Embedded	Wake-up CC
100	1423	611	609	608	2.622	2.622	2.622	2.622	0.007	0.005	0.005	0.005
200	2823	1211	1209	1208	2.724	2.622	2.622	2.622	0.009	0.006	0.006	0.006
300	4223	1811	1809	1808	2.724	2.622	2.622	2.622	0.011	0.007	0.006	0.007
400	5623	2411	2409	2408	2.827	2.622	2.622	2.622	0.013	0.007	0.007	0.007
500	7023	3011	3009	3008	2.827	2.711	2.724	2.724	0.016	0.008	0.007	0.008
600	8423	3611	3609	3608	2.929	2.724	2.724	2.724	0.017	0.008	0.008	0.008
700	9823	4211	4209	4208	2.929	2.724	2.724	2.724	0.019	0.009	0.008	0.009
800	11223	4811	4809	4808	3.032	2.724	2.724	2.724	0.022	0.009	0.009	0.01
900	12623	5411	5409	5408	3.032	2.724	2.724	2.827	0.024	0.010	0.009	0.01
1000	14023	6011	6009	6008	3.134	2.827	2.827	2.827	0.026	0.011	0.01	0.011

Table 2 Summary of sequential presentation complexity with varying number of processes – wake-up design

# of processes	# of States Stored	Total Actual Memory Used (Mb)	Real Time (sec)
3	1423	2.622	0.007
4	3047	2.724	0.011
5	6295	2.929	0.002
6	12791	3.339	0.044
7	25783	4.26	0.104
8	51767	6.104	0.241
9	103735	10.404	0.537
10	207671	19.211	1.268
11	415543	39.179	2.862
12	831287	79.012	6.589
13	1662780	168.817	15.384
14	3325750	348.632	40.789
15	6651700	747.582	118.032

Table 3 Summary of sequential presentation complexity with varying number of processes – the embedded, chain, and wake-up CC designs

# of do loops	# of States Stored			Total Actual Memory Used (Mb)			Real Time (sec)		
	Embedded	Chain	Wake-up CC	Embedded	Chain	Wake-up CC	Embedded	Chain	Wake-up CC
5	1013	1017	1012	2.622	2.622	2.622	0.006	0.006	0.005
10	2023	2032	2022	2.724	2.724	2.724	0.007	0.008	0.008
15	3033	3047	3032	2.827	2.827	2.929	0.009	0.011	0.012
20	4043	4062	4042	3.032	3.032	3.134	0.012	0.015	0.017
25	5053	5077	5052	3.236	3.339	3.441	0.013	0.019	0.024
30	6063	6092	6062	3.441	3.646	3.648	0.017	0.025	0.032
35	7073	7107	7072	3.748	3.953	4.260	0.020	0.032	0.042
40	8083	8122	8082	4.056	4.363	4.670	0.023	0.038	0.053
45	9093	9137	9092	4.465	4.875	5.284	0.027	0.048	0.064

Table 4 Comparison of Sequential Processes

Wake-up Model	**Embedded Model**

Starting :init: with pid 0
0: proc - (:root:) creates proc 0 (:init:)
Starting P1 with pid 1
1: proc 0 (:init:) creates proc 1 (P1)
1: proc 0 (:init:) line 51 "p4endstate100.txt" (state 5) [(run P1())]
Starting P2 with pid 2
2: proc 0 (:init:) creates proc 2 (P2)
2: proc 0 (:init:) line 51 "p4endstate100.txt" (state 2) [(run P2())]
Starting P3 with pid 3
3: proc 0 (:init:) creates proc 3 (P3)
3: proc 0 (:init:) line 51 "p4endstate100.txt" (state 3) [(run P3())]
Starting P4 with pid 4
4: proc 0 (:init:) creates proc 4 (P4)
4: proc 0 (:init:) line 51 "p4endstate100.txt" (state 4) [(run P4())]
5: proc 1 (P1) line 6 "p4endstate100.txt" (state 5) [((stateA1<endstate))]
6: proc 1 (P1) line 8 "p4endstate100.txt" (state 2) [stateA1 = (stateA1+1)]
7: proc 1 (P1) line 12 "p4endstate100.txt" (state 6) [.(goto)]
8: proc 1 (P1) line 6 "p4endstate100.txt" (state 5) [((stateA1<endstate))]
9: proc 1 (P1) line 8 "p4endstate100.txt" (state 2) [stateA1 = (stateA1+1)]
10: proc 1 (P1) line 12 "p4endstate100.txt" (state 6) [.(goto)]
------------- 	
depth-limit (-u10 steps) reached
#processes: 5
stateA1 = 2
stateA2 = 0
stateA3 = 0
stateA4 = 0
endstate = 100
10: proc 4 (P4) line 40 "p4endstate100.txt" (state 1)
10: proc 3 (P3) line 29 "p4endstate100.txt" (state 1)
10: proc 2 (P2) line 17 "p4endstate100.txt" (state 1)
10: proc 1 (P1) line 6 "p4endstate100.txt" (state 5)
10: proc 0 (:init:) line 53 "p4endstate100.txt" (state 6) <valid end state>
5 processes created
	Starting :init: with pid 0
	0: proc - (:root:) creates proc 0 (:init:)
	Starting P1 with pid 1
	1: proc 0 (:init:) creates proc 1 (P1)
	1: proc 0 (:init:) line 53 "1p6do.txt" (state 1) [(run P1())]
	2: proc 1 (P1) line 7 "1p6do.txt" (state 5) [((stateA1<endstate))]
	3: proc 1 (P1) line 9 "1p6do.txt" (state 2) [stateA1 = (stateA1+1)]
	4: proc 1 (P1) line 13 "1p6do.txt" (state 6) [.(goto)]
	5: proc 1 (P1) line 7 "1p6do.txt" (state 5) [((stateA1<endstate))]
	6: proc 1 (P1) line 9 "1p6do.txt" (state 2) [stateA1 = (stateA1+1)]
	7: proc 1 (P1) line 13 "1p6do.txt" (state 6) [.(goto)]
	8: proc 1 (P1) line 7 "1p6do.txt" (state 5) [((stateA1<endstate))]
	9: proc 1 (P1) line 9 "1p6do.txt" (state 2) [stateA1 = (stateA1+1)]
	10: proc 1 (P1) line 13 "1p6do.txt" (state 6) [.(goto)]
	------------- 	
	depth-limit (-u10 steps) reached
	#processes: 2
	stateA1 = 3
	stateA2 = 0
	stateA3 = 0
	stateA4 = 0
	stateA5 = 0
	stateA6 = 0
	endstate = 100
	10: proc 1 (P1) line 7 "1p6do.txt" (state 5)
	10: proc 0 (:init:) line 55 "1p6do.txt" (state 2) <valid end state>
	2 processes created

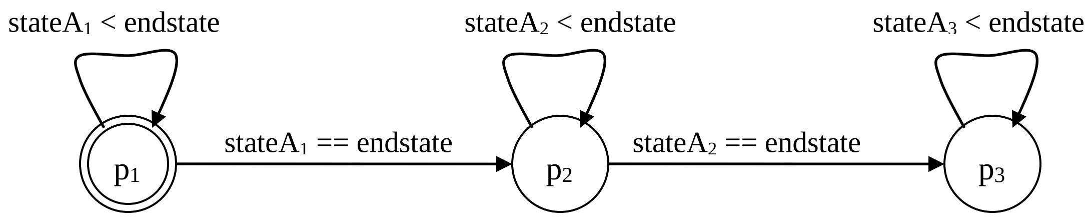
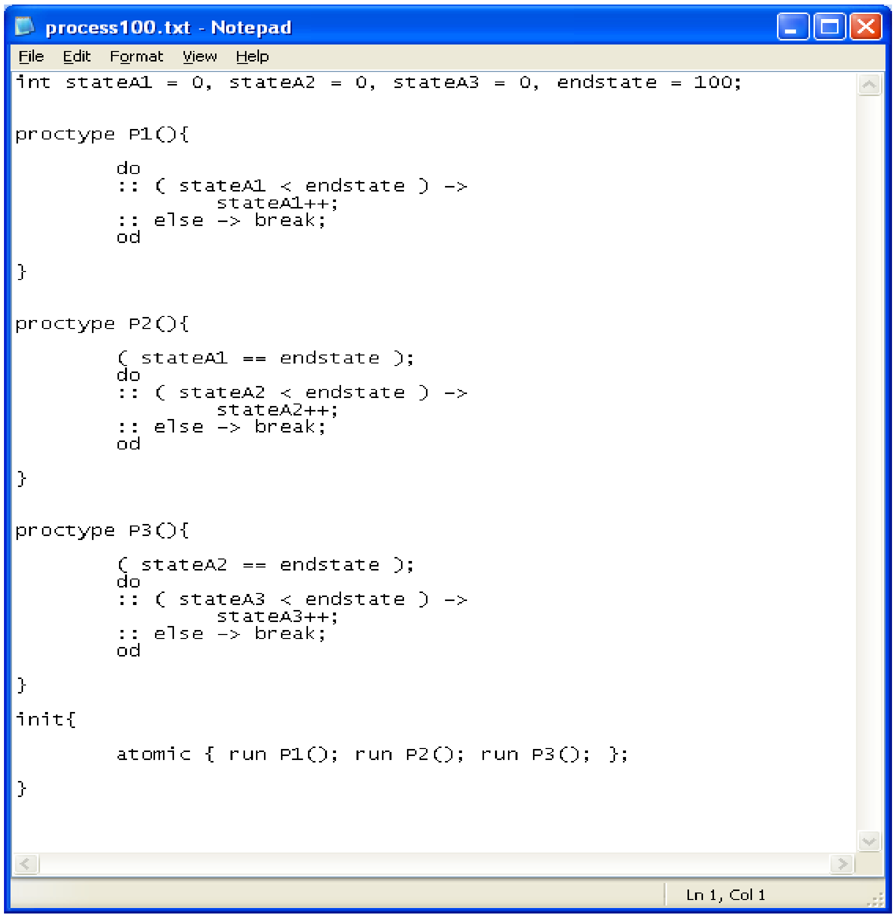
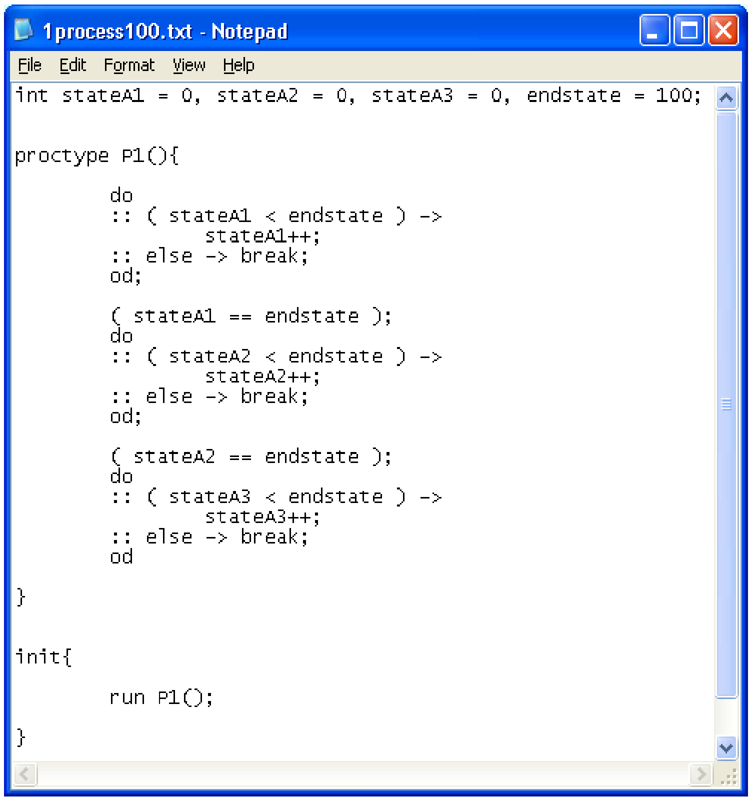


Figure 1 State Diagram for Do Loop Program Code Using Three Streams



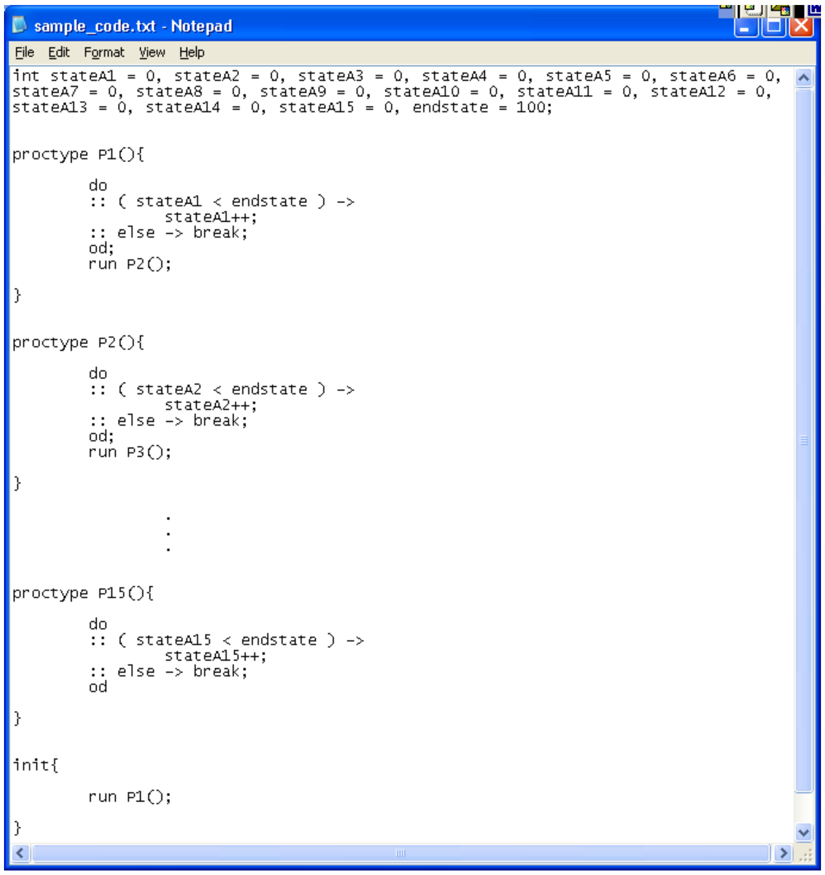
```
int stateA1 = 0, stateA2 = 0, stateA3 = 0, endstate = 100;

proctype P1(){

        do
        :: ( stateA1 < endstate ) ->
                stateA1++;
        :: else -> break;
        od

}

proctype P2(){

        ( stateA1 == endstate );
        do
        :: ( stateA2 < endstate ) ->
                stateA2++;
        :: else -> break;
        od

}

proctype P3(){

        ( stateA2 == endstate );
        do
        :: ( stateA3 < endstate ) ->
                stateA3++;
        :: else -> break;
        od

}
init{

        atomic { run P1(); run P2(); run P3(); };

}
```

Figure 2 Sample Promela code for the wake-up design having three streams



```
int stateA1 = 0, stateA2 = 0, stateA3 = 0, endstate = 100;


proctype P1(){

        do
        :: ( stateA1 < endstate ) ->
                stateA1++;
        :: else -> break;
        od;

        ( stateA1 == endstate );
        do
        :: ( stateA2 < endstate ) ->
                stateA2++;
        :: else -> break;
        od;

        ( stateA2 == endstate );
        do
        :: ( stateA3 < endstate ) ->
                stateA3++;
        :: else -> break;
        od

}


init{

        run P1();

}
```

Figure 3 Sample Promela code for the embedded design having three streams



```
int stateA1 = 0, stateA2 = 0, stateA3 = 0, stateA4 = 0, stateA5 = 0, stateA6 = 0,
stateA7 = 0, stateA8 = 0, stateA9 = 0, stateA10 = 0, stateA11 = 0, stateA12 = 0,
stateA13 = 0, stateA14 = 0, stateA15 = 0, endstate = 100;

proctype P1(){

        do
        :: ( stateA1 < endstate ) ->
                stateA1++;
        :: else -> break;
        od;
        run P2();

}

proctype P2(){

        do
        :: ( stateA2 < endstate ) ->
                stateA2++;
        :: else -> break;
        od;
        run P3();

}

                .
                .
                .

proctype P15(){

        do
        :: ( stateA15 < endstate ) ->
                stateA15++;
        :: else -> break;
        od

}

init{

        run P1();

}
```

Figure 4 Sample Promela code for the chain design having 15 streams

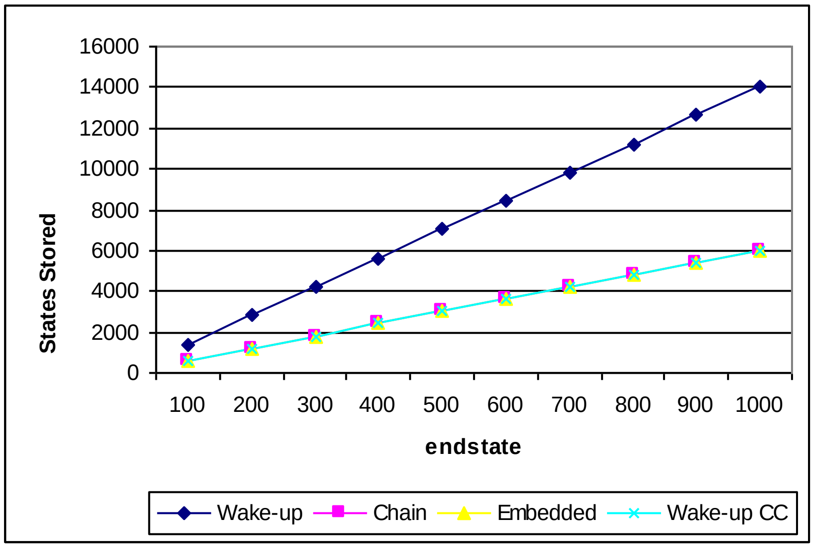


Figure 5 States stored (three streams)

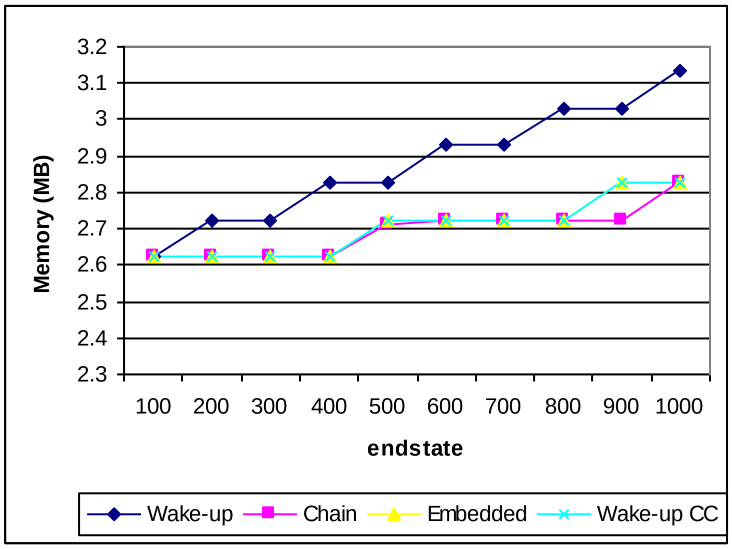


Figure 6 Memory usage (Mb) (three streams)

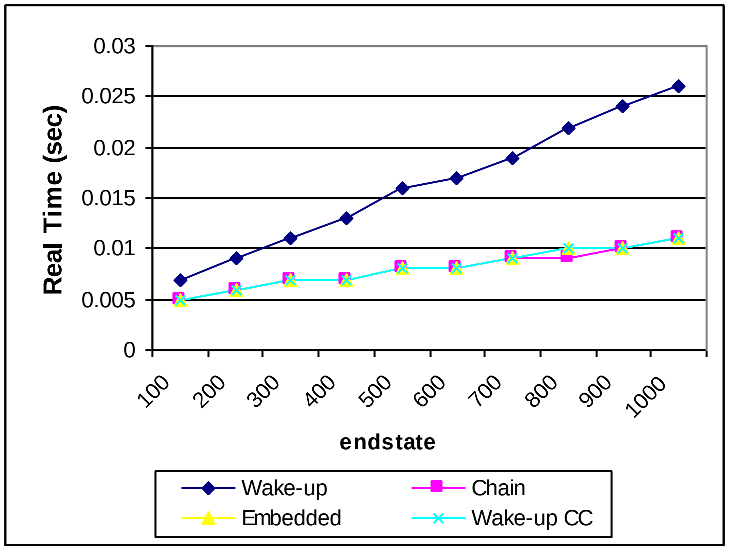


Figure 7 Real Time (sec) – all designs (three streams)

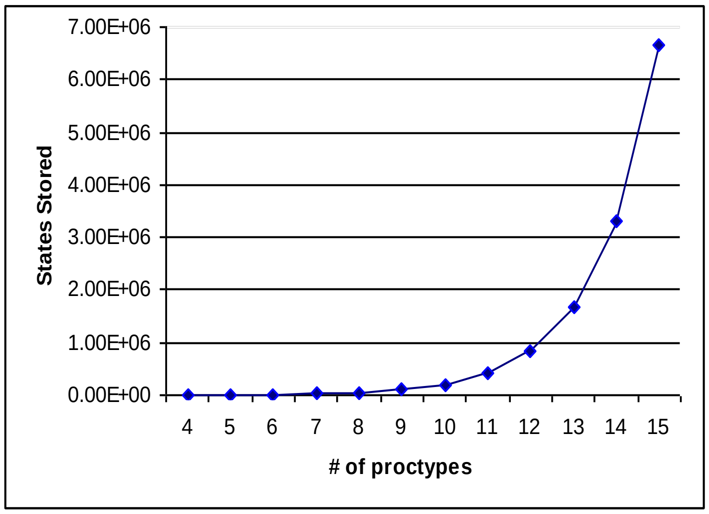


Figure 8 States stored – the wake-up design.

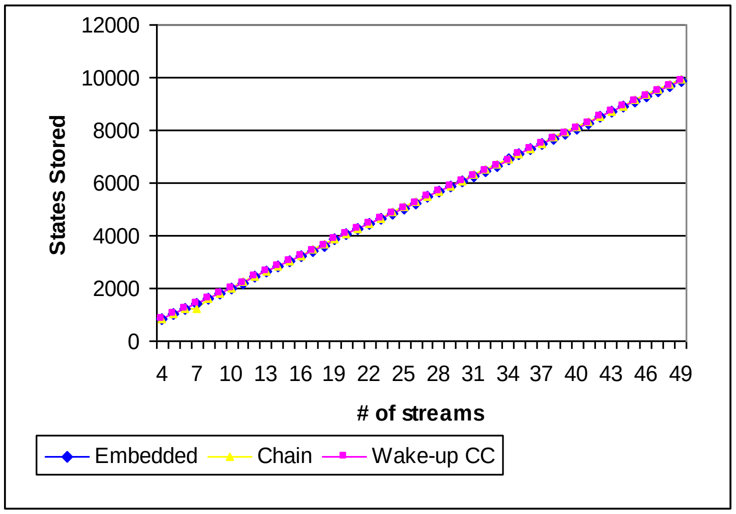


Figure 9 States stored – the chain, embedded, and wake-up CC designs.

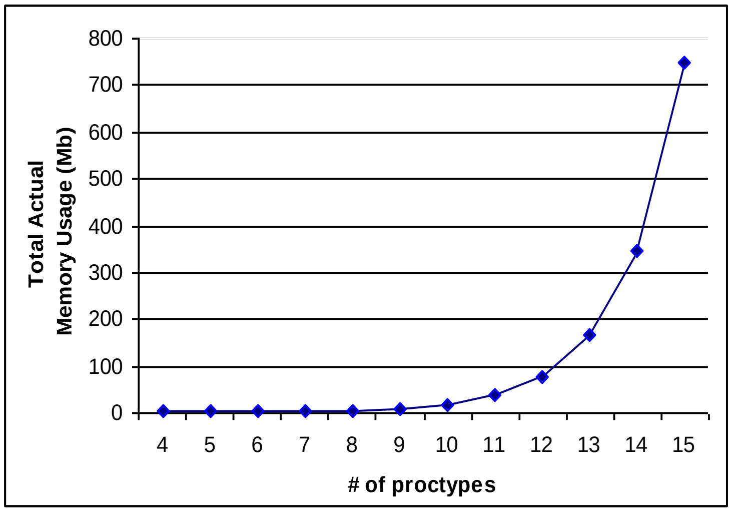


Figure 10 Memory usage (Mb) – the wake-up design

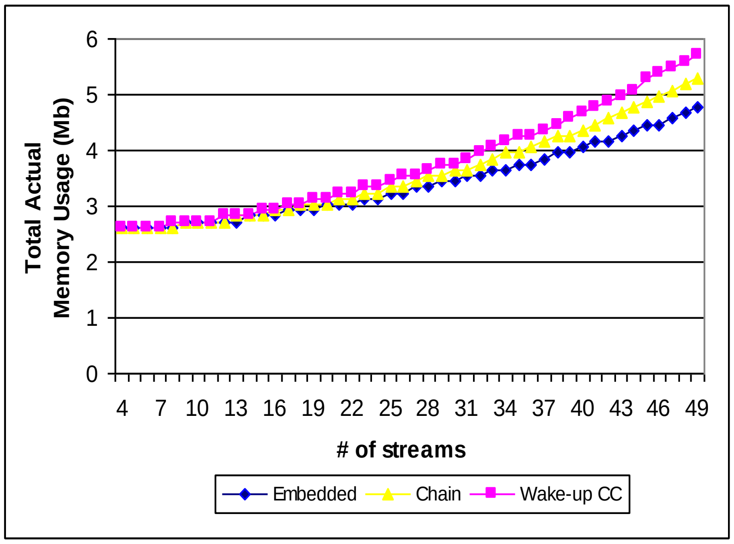


Figure 11 Memory usage (Mb) – the chain, embedded, and wake-up CC designs.

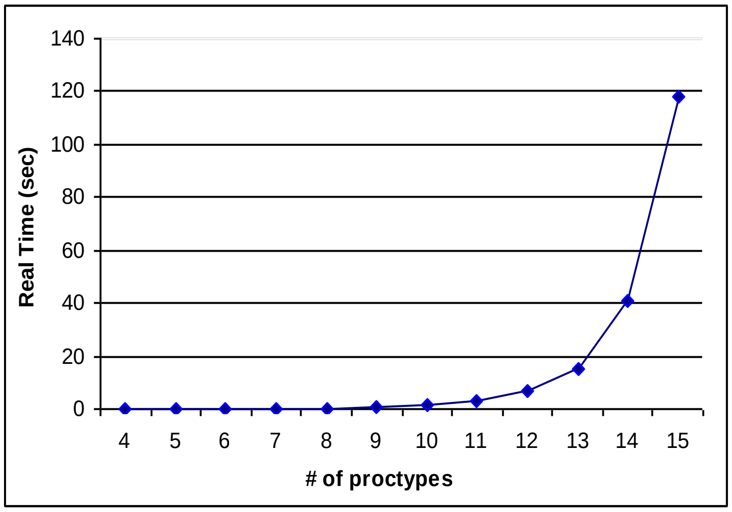


Figure 12 Real time (sec) – the wake-up design

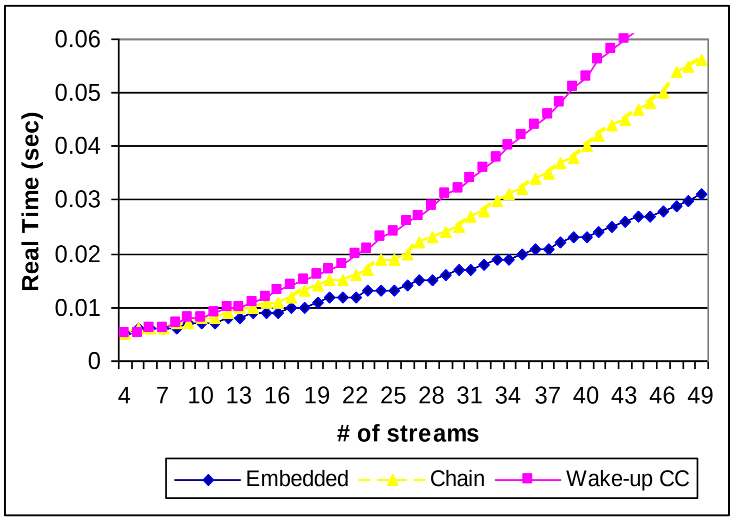


Figure 13 Real time (sec) – the chain, embedded, and wake-up CC designs.