Zing: Exploiting Program Structure for Model Checking Concurrent Software

Tony Andrews*, Shaz Qadeer*, Sriram K. Rajamani*,
Jakob Rehof*, and Yichen Xie†

*Microsoft Research
†Stanford University
http://www.research.microsoft.com/zing/

Abstract. Model checking is a technique for finding bugs in systems by systematically exploring their state spaces. We wish to extract sound models from concurrent programs automatically and check the behaviors of these models systematically. The ZING project is an effort to build a flexible infrastructure to represent and model check abstractions of large concurrent software.

To support automatic extraction of models from programs written in common programming languages, ZING's modeling language supports three facilities present in modern programming languages: (1) procedure calls with a call-stack, (2) objects with dynamic allocation, and (3) processes with dynamic creation, using both shared memory and message passing for communication. We believe that these three facilities capture the essence of model checking modern concurrent software.

Building a scalable model-checker for such an expressive modeling language is a huge challenge. ZING's modular architecture provides a clear separation between the expressive semantics of the modeling language, and a simple view of ZING programs as labeled transition systems. This separation has allowed us to decouple the design of efficient model checking algorithms from the complexity of supporting rich constructs in the modeling language.

ZING's model checking algorithms have been designed to exploit existing structural abstractions in concurrent programs such as processes and procedure calls. We present two such novel techniques in the paper: (1) compositional checking of ZING models for message-passing programs using a conformance theory inspired by work in the process algebra community, and (2) a new summarization algorithm, which enables ZING to reuse work at procedure boundaries by extending interprocedural dataflow analysis algorithms from the compiler community to analyze concurrent programs.

1 Introduction

The goal of the ZING project is to check properties of concurrent heap-manipulating programs using model checking. By systematically exploring the state space, model checkers are able to find tricky concurrency errors that are impossible to find using conventional testing methods. Industrial software has

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 1–15, 2004.

such large number of states and it is infeasible for any systematic approach to cover all the reachable states. Our goal is to automatically extract a *model* from a program, where a model keeps track of only a small amount of information about the program with respect to the property being checked. Then, it is feasible to systematically explore all the states of the model. Further, we want these models to be sound abstractions of the program —a property proved on the model should hold on the program as well.

How expressive should the model be? Choosing a very restricted model such as finite-state machines makes the task of building the model checker easy, but the task of extracting such a model from a program becomes hard. On the other hand, building a model checker directly for a programming language is hard, due to the number of features present in programming languages. We believe that the following features capture the essence of modern concurrent object oriented languages, from the point of building sound abstractions for model checking: (1) procedure calls with a call-stack, (2) objects with dynamic allocation, and (3) processes with dynamic creation, using both shared memory and message passing for communication. We designed ZING's modeling language to have exactly these features.

Building a scalable model checker for the ZING modeling language is a huge challenge since the states of a ZING model have complicated features such as processes, heap and stack. We designed a lower-level model called ZING object model (or ZOM), and built a ZING compiler to convert a ZING model to ZOM. The compiler provides a clear separation between the expressive semantics of the modeling language, and a simple view of ZOM as labeled transition systems. This separation has allowed us to decouple the design of efficient model checking algorithms from the complexity of supporting rich constructs in the modeling language.

Writing a simple DFS model checker on top of ZOM is very easy and can be done with a 10-line loop. However, this simple model checker does not scale. For building scalable checkers, we have to exploit the structural boundaries present in the source program that are preserved in the ZING model. Processes, procedures and objects are perhaps the structural abstractions most widely used by programmers. Structural boundaries enable compositional model checking, and help alleviate the state-explosion problem. For implementing optimized model checking algorithms that exploit such structure, we had to expose more information about the state of the model in ZOM.

In well-synchronized shared memory programs, any computation of a process can be viewed as a sequence of transactions, each of which appears to execute atomically to other processes. An action is called a right (left) mover if it can be committed to the right (left) of any action of another process in any execution. A transaction is a sequence of right movers, followed by at most a single atomic action, followed by a sequence of left movers. During model checking, it is sufficient to schedule processes only at transaction boundaries, and this results in an exponential reduction in the number of states explored. To implement such transaction-based reduction, we extended the ZOM to expose information about

the type of action executed —right mover, left mover, both left and right mover, neither left nor right mover.

The ability to summarize procedures is fundamental to building scalable interprocedural analyses. For sequential programs, procedure summarization is well-understood and used routinely in a variety of compiler optimizations and software defect-detection tools. This is not the case for concurrent programs. If we expose procedure boundaries in the ZOM, we can summarize procedures that are entirely contained within transactions. When a transaction starts in one procedure and ends in another, we can break the summary piece-wise and record smaller sub-summaries in the context of each sub-procedure. The procedure summaries thus computed allow reuse of analysis results across different call sites in a concurrent program, a benefit that has hitherto been available only to sequential programs [15].

We are interested in checking that a process in a communicating system cannot wait indefinitely for a message that is never sent, and cannot send a message that is never received. A process that passes this check is said to be *stuck-free* [16, 7, 8]. We have defined a conformance relation $\leq$ on processes with the following substitutability property: If $I \leq C$ and P is any environment such that the parallel composition $P \mid C$ is stuck-free, then $P \mid I$ is stuck-free as well. Substitutability enables a component's specification to be used instead of the component in invocation contexts, and hence enables model checking to scale. By exposing observable events during the execution of each action in ZOM, we can build a conformance-checker to check if one ZING model (the implementation) conforms with another ZING model (the specification).

The goal of this paper is to describe the architecture and algorithms in ZING. A checking tool is useless without compelling applications where the checker provides value. We have used ZING to check stuck-freeness of distributed applications, concurrency errors in device drivers, and protocol errors in a replicated file system. We have also built extractors from several programming languages to ZING. Since ZING provides core features of object-oriented languages, building such extractors is conceptually simple. Describing the details of these applications and extractors is beyond the scope of this paper.

To summarize, the ZING project is centered around three core principles:

1. It is possible to extract sound models from concurrent programs. To enable construction of simple extractors from common programming languages, the ZING modeling language has three core features (1) procedure calls, (2) objects and (3) processes.
2. It is beneficial to construct an intermediate model ZOM, which presents a simple view of ZING models as labeled transition systems. We have constructed various model checkers over this simple view.
3. Since ZING's modeling language preserves abstraction boundaries in the source program, we can exploit these boundaries to do compositional model checking, and help alleviate the state-explosion problem. Doing this requires exposing more information about the state and actions in ZOM. By exposing mover information about executed actions we have been able to imple-

ment transaction based reduction. By exposing information about procedure boundaries, we have been able to implement a novel summarization algorithm for concurrent programs. By exposing the observable events during execution of each action, we have been able to build a novel conformance checker to compositionally check if a ZING model is stuck-free.

Related Work. The SPIN project [10] pioneered explicit-state model checking of concurrent processes. The SPIN checker analyzes protocol-descriptions written in the PROMELA language. Though PROMELA supports dynamic process creation, it is difficult to encode concurrent software in PROMELA due to absence of procedure calls and objects. Efforts have been made to abstract C code into PROMELA [11] to successfully find several bugs in real-life telephone switching systems, though no guarantees were given as to whether the generated PROMELA model is a sound abstraction of the C code. Over the past few years, there has been interest in using SPIN-like techniques to model check software written in common programming languages. DSPIN was an effort to extend SPIN with dynamic software-like constructs [12]. Model checkers have also been written to check Java programs either directly [21, 20, 18] or by constructing slices or other abstractions [6]. Unlike ZING none of these approaches exploit program abstractions such as processes and procedure calls to do modular model checking. The SLAM project [4] has similar goals to ZING in that it works by extracting sound models from C programs, and checking the models. SLAM has been very successful in checking control-dominated properties of device drivers written in C. Unlike ZING, it does not handle concurrent programs, and it is unable to prove interesting properties on heap-intensive programs.

Outline. The remainder of the paper is structured as follows. We explain the features of ZING's modeling language, and discuss the modular software architecture of ZING in Section 2. We discuss the novel compositional algorithms of ZING in Section 3. Section 4 concludes the paper with a discussion of current status and future work.

2 Architecture

ZING's modeling language provides several features to support automatic generation of models from programs written in common programming languages. It supports a basic asynchronous interleaving model of concurrency with both shared memory and message passing. In addition to sequential flow, branching and iteration, ZING supports function calls and exception handling. New processes are created via asynchronous function calls. An asynchronous call returns to the caller immediately, and the callee runs as a fresh process in parallel with the caller. Primitive and reference types, and an object model similar to C# or Java is supported, although inheritance is currently not supported. ZING also provides features to support abstraction and efficient state exploration. Any sequence of statements (with some restrictions) can be bracketed as atomic. This is essentially a directive to the model checker to not consider interleavings with

other processes while any given process executes an atomic sequence. Sets are supported, to represent collections where the ordering of objects is not important (thus reducing the number of potentially distinct states ZING needs to explore). A choose construct that can be used to non-deterministically pick an element out of a finite set of integers, enumeration values, or object references is provided. A complete language specification can be found in [1]. An example ZING model that we extracted from a device driver, and details of an error trace that the ZING model checker found in the model can be found in [2].

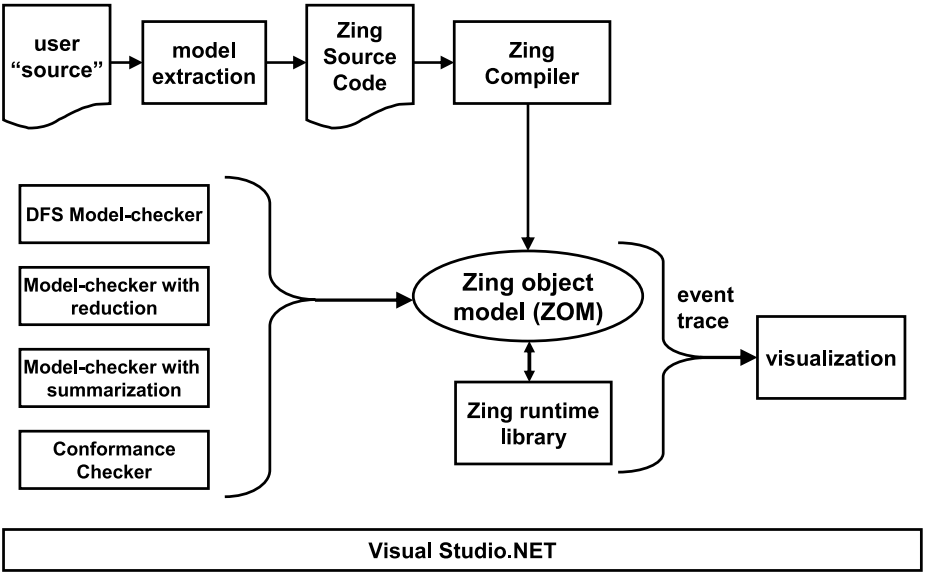

Fig. 1. Architecture of ZING

ZING is designed to have flexible software architecture. The architecture is designed to promote an efficient division of labor between model checking researchers and domain experts, and make it possible for model checking researchers to innovate in the core state-space exploration technology while allowing domain-experts to tackle issues such as extracting ZING models from their source code, and visualization for showing results from the model checker. Once model extraction is done, the generated ZING model is fed into a ZING compiler which converts the ZING model into an MSIL[1] object code called ZING object model (ZOM). The object code supports a specific interface intended to be used by the model checker. The ZOM assembly has an object of type *State* which has a

[1] MSIL stands for Microsoft Intermediate Language which is the instruction set for Microsoft's Common Language Runtime.

stack for each process, a global storage area of static class variables, and a heap for dynamically allocated objects. Several aspects of managing the internals of the State object can be done generically, for all ZING models. This common state management functionality is factored into a the ZING runtime library.

The equality operator for the State class is overridden to test equality using a "fingerprint" of the state with the following property: (1) If state s_1 is a symmetric equivalent of state s_2 then $\mathit{fingerprint}(s_1) = \mathit{fingerprint}(s_2)$, and (2) If $\mathit{fingerprint}(s_1) = \mathit{fingerprint}(s_2)$, then states s_1 and s_2 are equivalent with a high probability. Because states are compared frequently and the state vector is potentially large, the use of fingerprints is generally advantageous. Further, when all of the immediate children of a state have been generated, the full state representation may be discarded provided the fingerprint is retained. Two states are equivalent if the contents of the stacks and global variables are *identical* and the heaps are *isomorphic*. The fingerprinting algorithm for the State object first constructs a canonical representation of the state by traversing the heap in a deterministic order [12]. Thus, equivalent states have equal fingerprints. We observe that most state transitions modify only a small portion of the State object. The State object records an "undo-log" and uses it to reverse transitions, thereby avoiding cloning the entire state while doing depth-first search.

```
Stack dfsStack;
Hashtable stateHash;
void addState(State I) {
    if (!stateHash.Contains(I)) {
        stateHash.Add(I);
        dfsStack.Push(I);
    }
}

void doDfs(State initialState) {
    addState(initialImplState);
    while (dfsStack.Count ≥ 1) {
        State I = (State) dfsStack.Peek();
        State newI = I.GetNextSuccessor();
        if (newI != null)
            addState(newI);
        else
            dfsStack.Pop();
    }
}
```

Fig. 2. Simple DFS model checker for ZING

The State object exposes a `GetNextSuccessor` method that returns the next successor of the state. By iteratively calling this method, all successor states of the current state can be generated. Model checkers use the method `GetNextSuccessor` to execute a process for one atomic step. The execution semantics of the process, which includes complicated activities like process creation, function call, exceptions, dynamic memory allocation, are all handled by the implementation of `GetNextSuccessor` using support from the ZING compiler and runtime. Model checkers are thus decoupled from the intricate execution semantics supported by ZING. The actual implementation of the State object is

quite complicated since it has to represent stacks for each process, a global area and the heap. Using the interface provided by ZOM's State object, a simple depth-first search model checker for ZING can be written in less than ten lines as shown in Figure 2. The model checker stores finger prints of visited states in a hash table `stateHash`. When visiting each new state, the model checker first checks if the fingerprint of the new state is already present in the `stateHash`, and if present avoids re-exploring the new state. When the checker reaches an erroneous state, the entire trace that leads to the error is present in the model checker's DFS stack, and we can display the trace at the source level (this is omitted in Figure 2 for simplicity).

3 Algorithms

Since ZING's modeling language preserves abstraction boundaries in the source program, we can exploit these boundaries to do compositional model checking, and help alleviate the state-explosion problem. Doing this requires exposing more information about the state and actions in ZOM. By exposing mover information about executed actions, we have been able to implement transaction based reduction. By exposing information about procedure boundaries, we have been able to implement a novel summarization algorithm for concurrent programs. By exposing the observable events during execution of each action, we have been able to build a novel conformance checker to compositionally check if a ZING model is stuck-free.

3.1 Model Checker with Reduction

We have implemented a state-reduction algorithm that has the potential to reduce the number of explored states exponentially without missing errors. This algorithm is based on Lipton's theory of reduction [13]. Our algorithm is based on the insight that in well-synchronized programs, any computation of a process can be viewed as a sequence of transactions, each of which appears to execute atomically to other processes. An action is called a *right mover* if can be commuted to the right of any action of another process in any execution. Similarly, an action is called a *left mover* if can be commuted to the left of any action of another process in any execution. A transaction is a sequence of right movers, followed by a single (atomic) action with no restrictions, followed by a sequence of left movers. During state exploration, it is sufficient to schedule processes only at transaction boundaries. These inferred transactions reduce the number of interleavings to be explored, and thereby greatly alleviate the problem of state explosion. To implement transaction-based reduction, we augmented the `GetNextSuccessor` method so that it returns the type of the action executed (i.e., left mover, right mover, non mover or both mover), and the model checker uses this information to infer transaction boundaries.

3.2 Model Checker with Summarization

The ability to summarize procedures is fundamental to building scalable interprocedural analyses. For sequential programs, procedure summarization is well-understood and used routinely in a variety of compiler optimizations and software defect-detection tools. The *summary* of a procedure P contains the state pair (s, s') if in state s, there is an invocation of P that yields the state s' on termination. Summaries enable reuse—if P is called from two different places with the same state s, the work done in analyzing the first call is reused for the second. This reuse is the key to scalability of interprocedural analyses. Additionally, summarization avoids direct representation of the call stack, and guarantees termination of the analysis even if the program has recursion.

However, the benefit of summarization is not available to concurrent programs, for which a clear notion of summaries has so far remained unarticulated in the research literature. ZING has a novel two-level model checking algorithm for concurrent programs using summaries [15]. The first level performs reachability analysis and maintains an explicit stack for each process. The second level computes a summary for each procedure. During the reachability analysis at the first level, whenever a process makes a procedure call, we invoke the second level to compute a summary for the procedure. This summary is returned to the first level, which uses it to continue the reachability analysis. The most crucial aspect of this algorithm is the notion of procedure summaries in concurrent programs. A straightforward generalization of a (sequential) procedure summary to the case of concurrent programs could attempt to accumulate all state pairs (s, s') obtained by invoking this procedure in any process. But this simple-minded extension is not that meaningful, since the resulting state s' for an invocation of a procedure P in a process might reflect updates by interleaved actions of concurrently executing processes. Clearly, these interleaved actions may depend on the local states of the other processes. Thus, if (s, s') is an element of such a summary, and the procedure P is invoked again by some process in state s, there is no guarantee that the invoking process will be in state s' on completing execution of P. However, in well-synchronized programs, any computation of a process can be viewed as a sequence of transactions, each of which appears to execute atomically to other processes. Thus, within a transaction, we are free to summarize procedures. Two main technical difficulties arise while performing transaction-based summarization of procedures:

– Transaction boundaries may not coincide with procedure boundaries. One way to summarize such transactions is to have a stack frame as part of the state in each summary. However, this solution not only complicates the algorithm but also makes the summaries unbounded even if all state variables have a finite domain. Our summaries do *not* contain stack frames. If a transaction begins in one procedure context and ends in another procedure context, we break up the summary into smaller sub-summaries each within the context of a single procedure. Thus, our model checking algorithm uses a combination of two representations—states with stacks and summaries without stacks.

– A procedure can be called from different phases of a transaction —the pre-commit phase or the post-commit phase. We need to summarize the procedure differently depending on the phase of the transaction at the call site. We solve this problem by instrumenting the source program with a boolean variable representing the transaction phase, thus making the transaction phase part of the summaries.

Assertion checking for concurrent programs with finite-domain variables and recursive procedures is undecidable [17]. Thus, the two-level model-checking algorithm is not guaranteed to terminate. However, if all variables are finite domain and every call to a recursive procedure is contained entirely within a transaction, the two-level algorithm will terminate with the correct answer [15].

```
int g;
int baz(int x, int y){
  g = x+1;
}
```

Fig. 3. Small example to illustrate patterns and effects

Our implementation of the two-level model checking algorithm in ZING represents a summary as a *pattern* and *effect* pair, rather than a state pair. A pattern is a partial map from (read) variables to values, and an effect is a partial map from (written) variables to values.The ZOM supports summarization by exposing (1) whether the executed action is a procedure call or return, and (2) what variables are read and written during an action. Patterns and effects enable better reuse of summaries than state pairs. For example, consider the function `baz` from Figure 3. If `baz` is called with a state `(x=0,y=1,g=0)`, it results in state `(x=0,y=1,g=1)`. We represent a summary of this computation as a pattern `(x=0)` and an effect `(g=1)`. Thus, if `baz` is called with a state `(x=0,y=10,g=3)`, it still matches the pattern `(x=0)`, and the effect `(g=1)` can be used to compute the resulting state `(x=0,y=10,g=1)`. In contrast, if the summary is represented as a state pair `((x=0,y=1,g=0), (x=0,y=1,g=1))`, then the summary cannot be reused if `baz` were called at state `(x=0,y=10,g=3)`.

The model checker BEBOP [3] from the SLAM project represents summaries as state pairs. In order to illustrate the efficiency of reuse we present empirical comparison between ZING's implementation of summarization and BEBOP's implementation. Since BEBOP supports model checking of sequential programs only, we do the comparison with a parameterized set of sequential ZING models shown in Figure 4. Program $P(n)$ contains n global boolean variables `g1,g2,...,gn` and n procedures `level1,level2,...,leveln`. Figure 5 shows the running times for ZING and BEBOP for models $P(10), P(20), \ldots, P(100)$. Due to the use of patterns and effects for representing summaries, the ZING runtime for $P(n)$ scales linearly with n.

```
class BoolProg {
  static bool g1;
  static bool g2;
  ...
  static bool g<n>;

  activate static void main() {
    level1(true, true, true);
    level1(true, true, true);
  }
  static void level<i>(bool p1, bool p2, bool p3) {
    bool a,b,c;
    a = false;b = false;c = false;
    while(!a|!b|!c) {
      if (!a)
        a = true ;
      else if (!b)
        {a = false; b = true;}
      else if (!c)
        {a = false; b = false; c = true;}
      g<i> = false;
      level<i+1>(a, b, c);
      g<i> = true;
      level<i+1>(a, b, c);
      g<i> = false;
   }}}
```

Fig. 4. Template to evaluate summary reuse using patterns and effects

3.3 Conformance Checker

We are interested in checking that a ZING process cannot get into a state where it waits for messages that are never sent (deadlock) or has sent messages that are never received (orphan messages, for example, unhandled exception messages). We say, informally, that a processes is *stuck* if it cannot make any transition whatsoever, and yet some component of it is ready to send or receive a message. We say that a process is *stuck-free*, if it cannot transition to a stuck state.[2]

In order to check for stuck-freedom compositionally (one component at a time) for a system of communicating processes, we have defined a refinement relation $\leq$, called stuck-free conformance, which allows us to regard one ZING process as a specification of another. Stuck-free conformance is a simulation relation on ZING processes, which (i) is preserved by all contexts and (ii) preserves the ability to get stuck. From these properties it follows that, if P and Q are ZING processes such that $P \leq Q$, then for any process R, if $R \mid Q$ is stuck-free, then $R \mid P$ is stuck-free ($P \mid Q$ denotes the parallel composition of P and Q,

[2] We have formalized the notion of stuckness and stuck-freedom for transition systems in CCS [14], and we refer to [8, 7] for the precise definition of stuck-free CCS processes.

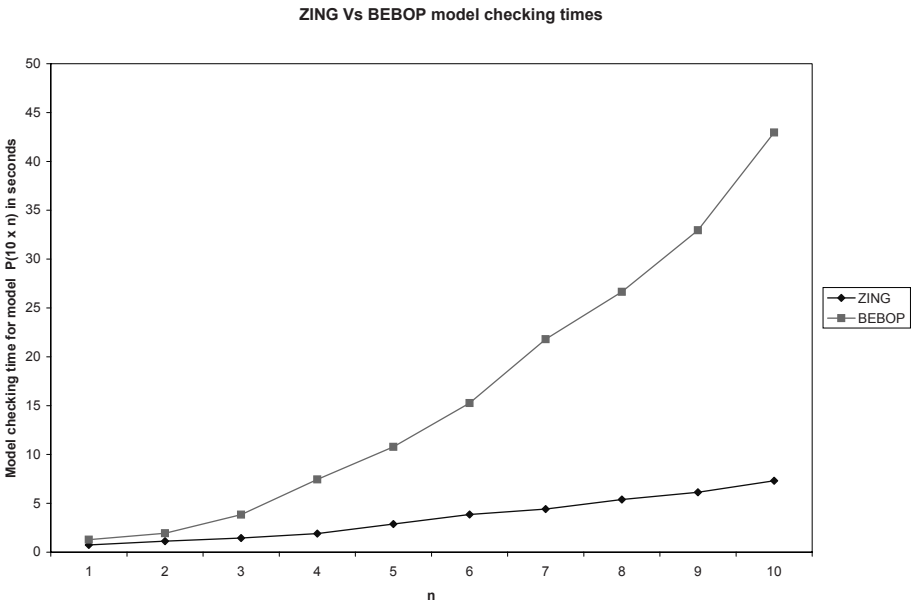

Fig. 5. Runtimes for ZING and BEBOP on models from Figure 4

which is expressed in ZING via `async` calls.) Therefore, if $P \leq Q$, we can safely substitute Q (a specification) for P (an implementation) in any context when reasoning about stuck-freedom, thereby enabling compositional checking. Our definition of stuck-free conformance [8, 7] between ZING processes is the largest relation $\leq$ such that, whenever $P \leq Q$, then the following conditions hold:

C1. If $P \xrightarrow{\tau^*\lambda} P'$ then there exists Q' such that $Q \xrightarrow{\tau^*\lambda} Q'$ and $P' \leq Q'$.
C2. If P can refuse X while ready on Y, then Q can refuse X while ready on Y.

Here, $P \xrightarrow{\tau^*\lambda} P'$ means that P can transition to P' on a sequence of hidden actions, τ, and a visible action, λ. A process is called *stable*, if it cannot do any τ-actions. If X and Y are sets of visible actions, we say that P *can refuse* X *while ready on* Y, if there exists a stable P' such that $P \xrightarrow{\tau^*} P'$ and (i) P' *refuses* X, i.e., P' cannot do a co-action of any action in X, and (ii) P' *is ready on* Y, i.e., P' can do every action in Y. In condition [C2] above, the ready sets Y range only over singleton sets or the empty set. This requirement on Y leads to the most permissive simulation satisfying the preservation properties (i) and (ii) mentioned above.[3]

We have extended the ZOM interface so that we can observe externally visible actions as well as the occurrence of hidden actions:

[3] Our notion of stuck-free conformance can be seen as a restriction of the natural simulation-based version of CSP stable failures refinement [5, 9, 19], which in addition to preserving deadlock also preserves the ability to generate orphan messages. We refer to [8, 7] for more details on the theory of stuck-free conformance.

```
Stack dfsStack;
Hashtable stateHash;
void addState(State I, State S) {
    StatePair combinedState = new StatePair(newI, newS);
    if (!stateHash.Contains(combinedState)) {
        stateHash.Add(combinedState);
        dfsStack.Push(combinedState);
    }
}

void checkConformance(State initialImplState, State initialSpecState) {
    addState(initialImplState, initialSpecState);
    while (dfsStack.Count ≥ 1) {
        StatePair P = (StatePair) dfsStack.Peek();
        State I = P.first();
        State S = P.second();
        State newI = I.GetNextSuccessor();
        if (newI == null) {
            if (isStable(I)) {
                // First get all the events we executed from I.
                ExternalEvent[] IEvents = I.AccumulatedExternalEvents;
                // Check if ready-refusals of I are ready-refused by S as well.
                for(int i = 0; i < IEvents.Count; i++) {
                    if(!checkReadyRefusals(S, IEvents, IEvents[i])) {
                        Console.WriteLine("Ready refusals do not match up");
                        return;
                    }
                }
            }
            dfsStack.Pop();
            continue;
        }
        ExternalEvent event = newI.ExternalEvent;
        // Try to produce a transition from newS with "event" as the observable event.
        State newS = executeWithEvent(S, event);
        if (newS == null) {
            Console.WriteLine("Implementation has unspecified behavior");
            return;
        }
        addState(newI, newS);
    }
    Console.WriteLine("I conforms with S");
}
```

Fig. 6. Conformance checker for ZING

1. `ExternalEvent` is a property which, for a newly generated state, gives the event (if any) on the transition that was used to generate the state.
2. `AccumulatedExternalEvents` gives an array of events from all outgoing transitions on a state, once all the outgoing transitions have been explored.

An implementation of the conformance checker using this interface is given in Figure 6. By exploring the state spaces of a given process P and a specification process C, `checkConformance`(P, C) decides whether $P \leq C$, by a direct implementation of conditions [C1] and [C2]. We assume that the specification does not have hidden nondeterminism. i.e., for a specification state S, if $S \xrightarrow{\tau^*.\lambda} S_1$ and $S \xrightarrow{\tau^*.\lambda} S_2$, then $S_1 \equiv S_2$. This assumption can be relaxed by determinizing the specification in a pre-processing step, or on-the fly using a subset construction. The conformance checker works by doing a depth-first-search on the state-space

of the implementation, and tracking the "matching" state of the specification corresponding to each state of the implementation. A hashtable is used to keep track of states that have been already visited. In our implementation, we store fingerprints of the visited states in the hashtables for efficiency. At each transition explored in the implementation, the algorithm checks for conditions [C1]. After all the successors of an implementation state have been explored, it is popped from the DFS stack. At that time, the algorithm checks if condition [C2] holds. The algorithm uses three functions `executeWithEvent`, `isStable`, and `checkReadyRefusals`. The function `executeWithEvent` searches the specification for a state which can be obtained by transitioning through the given event. Formally, `executeWithEvent`(S, λ) returns a state S' such that $S \xrightarrow{\tau^* \lambda} S'$ if such a state S' exists (note that such a state is unique if it exists due to the assumption that the specification does not have hidden nondeterminism). If this function returns null, then we conclude that condition [C1] has been violated. The function `isStable` returns TRUE if the given state S is stable and FALSE otherwise. The function `checkReadyRefusals`(S, X, λ) returns true if condition [C2] holds. More precisely, `checkReadyRefusals`(S, X, λ) returns TRUE if there exists a stable S' such that (i) $S \xrightarrow{\tau*} S'$ and (ii) for all λ' if $Q' \xrightarrow{\lambda'}$ then $\lambda' \in X$, and (iii) $S' \xrightarrow{\lambda}$. The algorithm terminates if the state space of the implementation is finite, and the complexity is linear in the state spaces of the implementation and the specification. If the state space of the implementation is too large or infinite, the algorithm can be used to check for conformance in whatever portion of the state space is explored.

4 Conclusion

The goal of the ZING project is to check properties of concurrent programs that manipulate the heap, by using natural abstraction boundaries that exist in the program. In order to support this goal, the ZING modeling language supports the essential features of modern object oriented languages, and the ZING architecture enables a clear separation between the expressiveness of the modeling language and the simplicity of the ZING object model (ZOM). This separation has enabled us to implement several novel model checking algorithms on top of the ZOM. We are currently implementing a few additional algorithms to enable ZING to scale to larger models:

- Currently non-determinism in data (introduced by the choose statement) is handled by an explicit case-split. We have designed a technique to handle such non-determinism symbolically. Our proposed algorithm adds symbolic fix-point computing capability to ZING, with the possibility of using widening to accelerate convergence.
- We are currently investigating how to design a SLAM-like iterative refinement loop inside ZING. SLAM handles pointers by doing an apriori alias analysis, and using predicates to refine the imprecision in alias analysis. We believe that directly handling pointers in the abstraction will scale better.

We have used ZING to check stuck-freeness of distributed applications [8, 7], concurrency errors in devicedrivers,[4] and protocol errors in a replicated file system.[5] Though a discussion of these applications is beyond the scope of this paper, all of the above algorithms and optimizations were driven by the need to make ZING scale on these applications.

Acknowledgments. We thank Jakob Lichtenberg and Georg Weissenbacher for their efforts in making ZING work inside the SLAM engine. We thank Tom Ball and Byron Cook for several discussions regarding this effort. We thank Vlad Levin for making the ZING UI display error traces in terms of the driver's C code. Abhay Gupta wrote ZING models of a large file-replication protocol. This effort helped uncover several bugs and performance bottlenecks in ZING and one serious bug in the protocol. We thank Tony Hoare and Cedric Fournet for working with us on the theory of stuck-free conformance.

References

1. Zing Language Specification – http://research.microsoft.com/zing.
2. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. Technical report, Microsoft Research, 2004.
3. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.
4. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.
5. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
6. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE 01: International Conference on Software Engineering*, pages 177–187. ACM, 2001.
7. C. Fournet, C. A. R. Hoare, S. K. Rajamani, and J. Rehof. Stuck-free conformance theory for CCS. Technical Report MSR-TR-2004-09, Microsoft Research, 2004.
8. C. Fournet, C.A.R. Hoare, S.K. Rajamani, and J. Rehof. Stuck-free conformance. In *CAV 04: Computer-Aided Verification*, LNCS. Springer-Verlag, 2000.
9. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
10. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
11. G.J. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.
12. R. Iosif and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN 99: SPIN Workshop*, LNCS 1680, pages 261–276. Springer-Verlag, 1999.

[4] Jakob Lichtenberg and Georg Weissenbacher started this work as an intern project in the summer of 2003.

[5] Abhay Gupta did this work as an intern project in the summer of 2003.

13. R. J. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717–721, 1975.
14. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
15. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL 04: ACM Principles of Programming Languages*, pages 245–255. ACM, 2004.
16. S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 166–179. Springer-Verlag, 2002.
17. G. Ramalingam. Context sensitive synchronization sensitive analysis is undecidable. *ACM Trans. on Programming Languages and Systems*, 22:416–430, 2000.
18. Robby, M. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *FSE 03: Foundations of Software Engineering*, pages 267–276. ACM, 2003.
19. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
20. S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, October 2002.
21. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ICASE 00: Automated Software Engineering*, pages 3–12, 2000.