Spring 2010

Deadlock detection and dihomotopic reduction via progress shell decomposition

David Andrew Cape

Department: Computer Science

DEADLOCK DETECTION AND DIHOMOTOPIC REDUCTION VIA PROGRESS SHELL DECOMPOSITION

by

DAVID ANDREW CAPE

A DISSERTATION

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

2010

Approved by

Dr. Bruce McMillin, Advisor
Dr. Xiaoqing (Frank) Liu
Dr. Daniel Tauritz
Dr. Thomas Weigert
Dr. Sahra Sedigh

ABSTRACT

Deadlock detection for concurrent programs has traditionally been accomplished by symbolic methods or by search of a state transition system. This work examines an approach that uses geometric semantics involving the topological notion of dihomotopy to partition the state space into components, followed by an exhaustive search of the reduced state space. Prior work partitioned the state-space inductively; however, this work shows that a technique motivated by recursion further reduces the size of the state transition system. The reduced state space results in asymptotic improvements in overall runtime for verification. Thus, with efficient partitioning, more efficient deadlock detection and eventually more efficient verification of some temporal properties can be expected for large problems.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my research advisor, Dr. Bruce M. McMillin for support and guidance during this dissertation project. He has given valuable input on all aspects of the project. In addition, he provided superlative logistical support by assigning assistants to help with parts of the research and by providing equipment and laboratory space for the research. Dr. Mayur Thakur was my research advisor initially: he helped with experimental design and analysis of the results. I would like to thank supervisory committee members – Dr. Thomas Weigert, Dr. Xiaoqing (Frank) Liu, Dr. Daniel Tauritz, and Dr. Sahra Sedigh – who have provided input during this project. Overall, the Computer Science Department of Missouri S&T facilitated a safe yet interesting learning experience, for which I am also thankful.

Many others have contributed to the success of this research as well. Valuable feedback was received from conferences concerning the preliminary results. I am also grateful for the insight gained from correspondence with Drs. Eric Goubault, Martin Raussen, and Dr. Gerard Holzmann. Benjamin W. Passer helped with many of the first, most complex, and most rigorous geometric proofs of the correctness properties of the algorithms. Stephen C. Jackson implemented most of the modifications to SPIN, created the DRUiDD software from a high-level description that I provided, and offered important contributions to the design of the code. Colleagues in the office have frequently provided technical expertise, and my parents and friends have been extremely supportive as well.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

1.1. SOFTWARE

Like any other engineering discipline, software engineering exists primarily to construct a product. However, the software product is more malleable than most other engineering products [1]; it can be modified more easily than a manufactured product (although modifications may be time-consuming and expensive). In computer science, one can develop tools and techniques to assist in the software engineering process, often by automating a process or developing a better methodology for it. The need for costly software modifications can be reduced with accurate specifications and subsequent verification that the product meets those specifications. Such accuracy can also prevent accidents when the software is to be used for a purpose that involves human safety, such as control of transportation or generation and distribution of electric power.

Verifiability can increase confidence in the correctness of software; this feature is the main concern of this dissertation project. Verification confirms the software has the properties indicated in the product specifications. If the specifications are inaccurate or incomplete, however, the product may not satisfy customers. Still, analysis of the requirements and specifications is an important part of the software development process.

The traditional software development model has four stages: requirements analysis, program design, code implementation, and testing. Requirements analysis involves gathering and organizing the specifications. Program design can involve extensive modeling of objects, processes, and data flow. This step may also demand verification of the design, because design flaws can be a source of costly errors if identified late in the development process. Implementation is the actual programming work that translates the design into a language understandable by a computer. Finally, testing is intended to identify and eliminate any defects that may have arisen at earlier stages. Variations in this model have included more feedback at various stages, or cycling of stages (as in the spiral model); these modifications are appropriate when sufficient resources are available.

1.2. VERIFICATION

The level of tolerance for error in software depends largely on the intended application; small errors can have great consequences in some settings, but they may only be a minor nuisance in others. When tolerance for error is low, either because the specifications are narrowly targeted for a particular application or because the potential cost of error is great, there is a greater need for verification. And verification can be useful at other stages of development as well, particularly after implementation, although the process is more complex at this point.

One may ask why testing is not sufficient, since it is supposed to identify defects. Testing does not prove correctness; it can only prove faultiness or increase the confidence in certain functional aspects of the software. To increase the confidence in the fundamental design of the software, verification is necessary through one of various methods.

There are two well-known static analysis methods for software. The first relies on a human being to prove the correctness of system properties. The second relies on a computer to perform automatic verification. The first method relies on Hoare logic, while model checking is the basis for the second. Model checking software takes as input a state transition system in conjunction with a property expressed in some form of temporal logic.

Hoare logic, invented by C.A.R. Hoare, is based on the idea of attaching pre- and post-conditions to program statements in such a way as to ensure that larger-scale properties (e.g., loop invariants) hold. The conditions are written in Boolean formulas, and inference rules are given for linking them together. This method of verification can be slow and may require human expertise; therefore, the verification method presented here focuses instead on automatic verification by model checking software.

1.3. TEMPORAL LOGIC

Temporal logic has its formal origin in ancient Greece, where it was developed by Megarian and Stoic philosophers [2]. These philosophers were concerned with the meaning of the words possibly and necessarily, and they proposed that their meanings could be understood in terms of whether a condition must eventually occur (in which case it is possible) or whether it must occur always (in which case it is necessary). The Megarians sought to quantify over the past, present, and future, whereas the Stoics were concerned

with the present and future only. Computer scientists have adopted the Stoic point of view.

In computer science, the state of a program is understood to be the values of the important variables that would need to be saved if the program were to be interrupted and restarted, including the values of all data and control variables, and the contents of all message channels if applicable. A transition is a change from one program state to another that occurs without visiting any intermediate state. States and transitions may be represented by a directed graph called a *state transition system*, in which the vertices are the states and the directed edges are the transitions.

A state formula is a Boolean expression involving the values of the state variables. Any given state may be tested to determine whether it satisfies a particular state formula. Temporal formulas are more complicated because they express properties of the actual execution of a program through time. A temporal formula is developed by applying to state formulas temporal operators that capture the notion of eventuality, persistence, or some shift of context to the next or previous state. Some temporal operators express the idea of a property being true until another property becomes true, or of a property being true since another property was most recently true.

The two main types of temporal logic are linear-time temporal logic (LTL) and computation tree logic (CTL). The two are fairly similar except that LTL formulas are evaluated on single-execution traces, whereas CTL formulas are evaluated on the entire forward-branching computation tree, which expresses the range of possible executions of a program. Thus, CTL operators also specify the scope of paths through the computation tree to which they apply. For example, the CTL operator AG means *on all paths it is true from now on*. In LTL, one uses the box operator to express something similar to this (the box operator means *it is true from now on*). Manna and Pnueli [3] provide a complete exposition. Clarke [4] is a good reference for CTL.

1.4. MODEL CHECKING

SPIN is one of the most popular model checkers; it has been used for a number of verifications found in model checking. It is well-documented and has a fairly large

following. There is even a workshop devoted to its use and development, and it has been enhanced for a variety of purposes.

PROMELA (an acronym for process meta-language) is the modeling language for the code used by SPIN [5]. It incorporates features of CSP [6] and guarded commands [7]. Its data structures include ordinary numeric variables, arrays, and first-in-first-out (FIFO) message channels. The basic statements of PROMELA include simple assignments, nondeterministic selection constructs, and nondeterministic loop constructs. Each statement has one or more control points that represent stopping points between transitions in the execution of a program. Such control points are sufficient for sequential programs, but concurrent programs require a process structure. With concurrency also comes the addition of message-passing statements, and communication can be either rendez-vous or buffered. SPIN supports concurrency, and PROMELA permits the writing of concurrent processes, the transitions of which are interleaved nondeterministically.

The support of concurrency brings up the issue of deadlock of processes. SPIN can check ordinary assertions about the program state as the program executes, but it can also check for the possibility of deadlock in the execution of a program. SPIN takes as input a concurrent or sequential program written in PROMELA and explores all possible executions of the program to verify desired properties. In fact, it can even allow premature termination at specified valid end-states of individual processes without signaling deadlock. Another type of verification is the non-existence of non-progress cycles, which means that in any recurrent cycle of a program a progress state is visited at least once, guaranteeing that progress states are visited infinitely often in any infinite execution of a program.

Related to progress verification is the verification of LTL temporal formulas by means of "never claims." A never claim is a type of monitoring process which executes along with the concurrent program being checked and indicates whether the program satisfies or violates a given temporal assertion. SPIN verifies temporal formulas by negating them and converting them to a never claim monitor automaton. The program itself is converted to an automaton also, and SPIN checks for the emptiness of the intersection of the languages accepted by the program automaton and the never claim automaton. That is, it

verifies that the program never executes in a way that coincides with the execution of the automaton representing the negation of the temporal formula.

In addition to modeling, one must decide which properties to try to verify and determine how to represent those properties in a manner that permits verification by the model checker. A model checker, then, is a software package that takes as input a model of a (usually concurrent) program and some logical property or properties to be verified for the execution of the model. These properties are usually divided into safety (something bad never happens) or liveness (something good eventually happens) properties. Another decomposition of temporal properties is safety and progress (which is slightly more complicated to define; see [3]). The model checker can verify a property mainly by exhaustive search (as SPIN does) or by symbolic (fixed point) methods (as SMV does) [4, 8].

1.5. OPTIMIZATION

A limiting parameter in model checking is the size of the state transition system (of the concurrent program) to be explored. The state space explosion problem is due to the exponential increase in the number of paths through a state transition system as the size of the system itself increases. Model checkers must optimize the verification algorithms to address the state-space explosion problem, which makes the search space for these verifications very large due to nondeterminism in the execution of the individual processes and in the interleaving of multiple processes. One method to combat this problem is partial-order reduction, which finds a subspace of the state transition system that faithfully represents the whole system for the purposes of verification. Another method is state space minimization, which stores the state space efficiently in a special data structure similar to an ordered binary decision diagram (OBDD) [8]. The idea behind OBDDs is that one can conserve space by avoiding duplication of subtrees in a binary decision tree representing a Boolean function.

This work investigated an alternative method based on the topology and geometry of the state space of concurrent programs. By partitioning the state space into cuboidal components and exploring a reduced digraph, one can obtain a correct verification outcome

for deadlock detection. This optimization is possible because the reduced digraph captures both the essential topological features of the original state space and the geometric property of maximality (which is the essence of deadlock). In addition, this geometric perspective could conceivably lead to more efficient model checking of a more general class of LTL properties (i.e., the ones that are consistent on paths representing trace executions).

2. STATE OF THE ART

2.1. SYMBOLIC TECHNIQUES

To understand how symbolic model checking works, one must understand how to encode a state transition system as an OBDD. This encoding is done by forming the disjunction of formulas that represent transitions between states (or by using some higher-level formulation). Clarke [4] provides a simple example in his chapter on symbolic model checking. One set of variables encodes current states (i.e., starting points for transitions) and another encodes the values of those variables in the next state after a particular transition (i.e., endpoints of the transition). Therefore, a transition can be written as a conjunction of the formula describing the current state and of that describing the next state (using the second set of variables).

Basic CTL operators on formulas that describe current properties, not state spaces, can be evaluated as the least or greatest fixpoint of a monotonic function particular to the formula that operates on subsets of the state space. For example, let ϕ be a predicate on the state-space so that it is equivalent to the subset of the state space that satisfies ϕ. Then AF ϕ means the set of states s for which it is true that on any future path p starting at s, there exists at least one state s' in p such that s' satisfies ϕ. In other words, no matter which path is taken, ϕ is eventually true at some future state following s. Thus, AF ϕ is equal to the subset of the state space that is the least fixpoint of an operator on the lattice of subsets of the state space. The operator is written here as $\phi \vee$ AX. It takes a subset S of the state-space to another subset S' according to the rule: s' is in S' if s' satisfies ϕ or if all immediate successors of s' are elements of S (see Figure 2.1).

The proof that this representation of the CTL operators is valid is difficult because the notation is not easy to digest; Clarke [4] provides additional details. A procedure can be written to evaluate a CTL formula (not just a Boolean *state* formula, but one involving the CTL temporal operators as well) on the OBDD representing the state transition system, which is what model checkers are supposed to do.

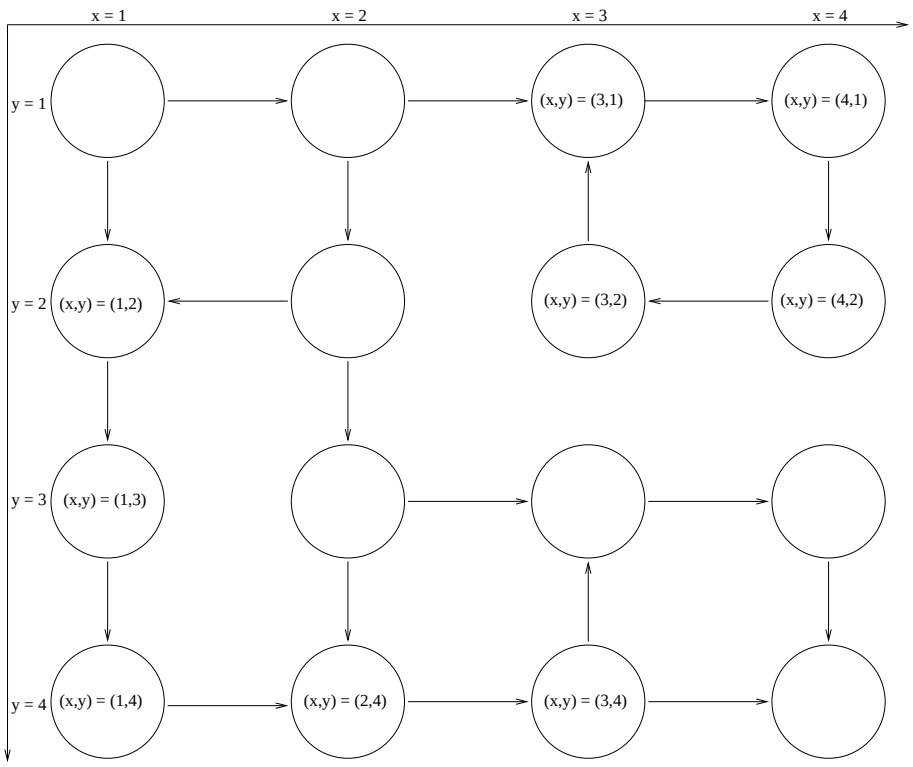

Figure 2.1. Let $\phi = [(x,y)=(3,4)]$. Then AF $\phi = \{(1,2), (1,3), (1,4), (2,4), (3,4)\}$. However, $\phi \vee$ AX has another fixpoint consisting of $\{(1,2), (1,3), (1,4), (2,4), (3,4)\} \cup \{(3,1), (4,1), (4,2), (3,2)\}$ that is not minimal.

2.2. PARTIAL-ORDER REDUCTION

The model checker SPIN uses a more traditional method of reducing the size of the state space, namely pruning transitions that do not affect the outcome of the verification. That is, given a property to be verified, one can try to find a subset of the state transition system of the concurrent program such that the property will be true for the subspace if and only if it is true for the original (larger) state transition system. The strategy involves eliminating transitions that correspond to interleavings of statements which are equivalent in some sense to a representative interleaving. It is as if one explores only one path from a given state to a given future state instead of exploring all possible paths to it. This technique is a kind of "local" version of dihomotopy, according to Goubault and Raussen [9].

2.3. SEARCH HEURISTICS

Model checking is, on the one hand, a search problem, for if one can quickly find a counterexample to a given property, the verification can be terminated early. On the

other hand, if the property happens to be true, early termination is impossible; rather, an exhaustive exploration of the paths through the state space is usually necessary for verification of software properties. (Hardware may be efficiently verified by symbolic methods; however, the present work is more concerned with model checking software.) Gradara, Santone, and Villani [10] have proposed a type of A* search based on a heuristic derived from process algebra, but this only improves the efficiency of finding a counterexample, not the efficiency of proving correctness by showing that counterexamples do not exist.

2.4. DIHOMOTOPY

This section introduces a modification of the deadlock detection algorithm described by Goubault et al. Goubault's model of a concurrent program works well for simple sequential processes (without branching or looping) by assigning each process an axis and modeling execution of the concurrent program by a path through the grid spanned by the axes (i.e., a progress graph). The topological notion of dihomotopy permits formation of equivalence classes of paths based on their Mazurkiewicz traces [11] (see Figure 2.2). A Mazurkiewicz trace is an equivalence class strings of transitions (in this context): two strings s and t represent the same trace *if and only if* there is a finite sequence of transformations from one string, s_i, to the next, s_{i+1}, beginning with $s_0 = s$ and ending with $s_n = t$, such that each transformation consists of transposing two consecutive independent elements of the string s_i. For example, suppose that there are four transitions in the alphabet, $\{a, b, c, d\}$, and $\{a, b\}$ is an independent set and $\{c, d\}$ is also independent. Then the equivalence class of the string *babdcdcabad* contains all strings obtainable from commuting the letters a and b and from commuting the letters c and d. Its simplest representative alphabetically is *abbccddaabd*, because *bab* is equivalent to *abb*, *dcdc* is equivalent to *ccdd*, and *aba* is equivalent to *aab*. Two transitions in a concurrent program can be transposed if neither one disables the other and if they lead to the same state no matter which of them is executed first.

A progress graph can be understood as a state space through which a (multiprocess) program takes a discrete sequence of states. Different interleavings result in different state sequences (paths). In general, a program may take many paths, and each path must be considered in program verification. For a concurrent program having linear processes (i.e.,

without branching or iteration), an alternate approach is to consider a progress graph through a geometric interpretation by assigning to the processes' axes. A two-process program, for example, begins at the origin (lower left) and moves towards the terminal point (upper right) of a geometric region. There may be many ways of getting from the initial point to the terminal point; these correspond to the possible paths through the progress graph. Synchronization primitives provide areas within the geometric region that are impossible to reach or are forbidden. Certain interior rectangles, corresponding to the forbidden regions, are removed. Conceptually, these rectangles represent areas that mutual exclusion will prevent the program state from entering. Consider a geometric region defined by the execution of two processes, with a semaphore guarding a single critical section in each process. The product of the two intervals corresponding to the critical sections forms a forbidden rectangle. In this interpretation, paths in the progress graph move from the lower left corner to the upper right corner, going around the forbidden region (or regions if the program has multiple critical sections; Figure 2.2). All paths that go around the forbidden regions in a certain way are equivalent – but they are different from paths that go around the forbidden regions in a different way.

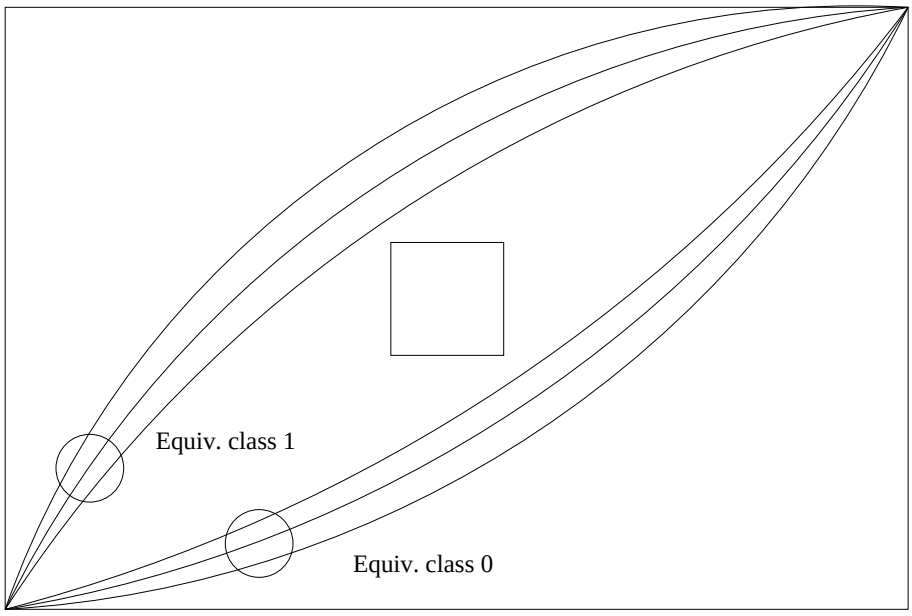

Figure 2.2. All paths that go above the interior forbidden square region are equivalent, as are all paths that go below it.

Formalizing this approach, Carson and Reynolds [12] modeled a concurrent program using geometric semantics; Pratt [13] has advocated the relevance of "higher-dimensional automata" to the study of concurrent programs; and Eric Goubault [14] and his collaborators [9, 15, 16] have introduced dihomotopy to represent equivalence of paths by continuous deformations between them. As more semaphores are added to the processes, the geometry becomes more complex and more nonequivalent paths arise [17]. Carving up this state-space and merging states into as few regions as possible is an attractive strategy for coping with the complicated topology. In this way, paths simply transition between these regions: the fewer the regions, the fewer the paths. Goubault's prior work [15] presented an inductive method of decomposing a progress graph into a collection of such aggregate states (regions). This work proposes a method that considers all forbidden rectangles together from the start.

When locking and unlocking semaphore statements and mutual exclusion force paths to go around forbidden regions of the grid, there can be more than one trace. In general, the term *progress graph* is used to refer to the continuous topological space obtained by filling in hypercubes between grid-points and then removing hypercubes corresponding to regions forbidden by the semaphore rules. In this work, all semaphores are binary. Figure 2.3 depicts a forbidden rectangle, and Figure 2.4 depicts the Swiss flag progress graph.

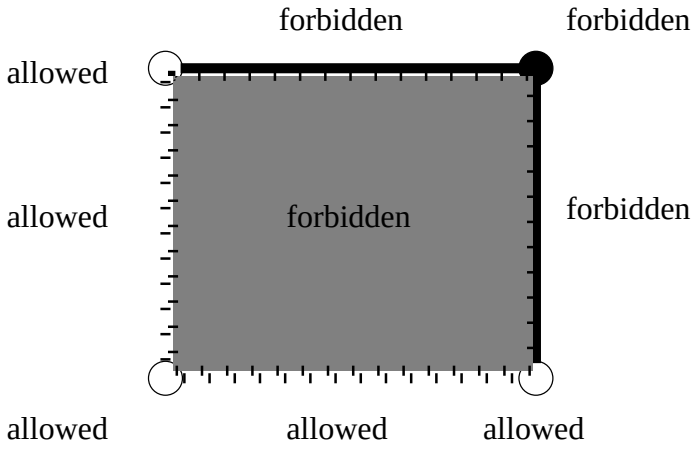

Figure 2.3. An example of a forbidden rectangle illustrating the fact that some of the boundary is actually permitted.

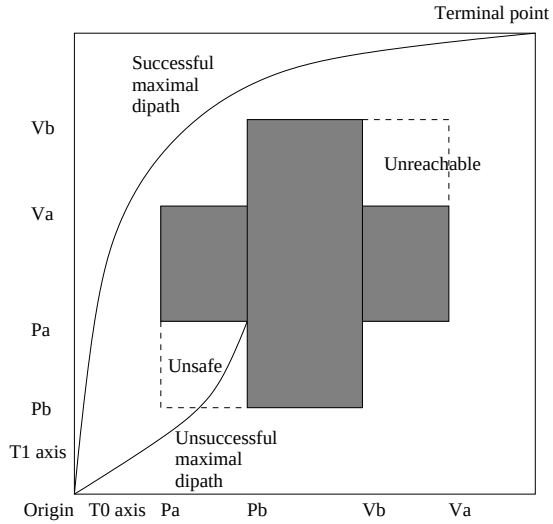

Figure 2.4. This is the standard two-semaphore Swiss flag example. Two processes, $T_0 = \{Pa; Pb; Vb; Va;\}$ and $T_1 = \{Pb; Pa; Va; Vb;\}$, comprise a concurrent program. The horizontal rectangle corresponds to a, and the vertical rectangle corresponds to b; they are forbidden. The minimal corners of each rectangle occur at points labeled (Pa, Pa) and (Pb, Pb). Likewise, the maximal corners occur at points labeled (Va, Va) and (Vb, Vb).

Because the execution of a linear process is always "forward-moving" and never returns to any control point that it has left, the only paths allowed are those that are nondecreasing in each direction specified by an axis. Certain areas of the progress graph may be unsafe, and certain areas may be unreachable. If an unsafe region is reachable, then deadlock is possible for the program modeled by the progress graph. This possibility is illustrated for the Swiss flag example by the execution in which the first process locks its first semaphore and the second process locks the other semaphore.

The Swiss flag example is the simplest type of dining philosopher problem, in which two philosophers each vie for two forks or chopsticks. In general, there are N philosophers and N utensils arranged in a circle so that each utensil is between two philosophers and each philosopher is between two utensils so that the philosophers and utensils alternate around the circumference of the circle [6]. Deadlock occurs when each philosopher picks up one utensil and refuses to release it because each philosopher needs both utensils to eat.

Dijkstra originally used progress graphs with axes corresponding to the processes [12, 18]. Goubault and others [9, 15], however, have advanced the concept of using higher-dimensional automata [13, 14] and dihomotopy to simplify the analysis of progress graphs by decomposing them into components. The larger the components, the fewer there will be, and the greater the reduction of the state space explosion.

Partitioning a progress graph into components that are equivalence classes of states can reduce the size of the state space to be explored and therefore reduce the number of paths to be checked for any given property. For problems with many semaphores per process, a partitioning method like Goubault's that uses a modified partitioning technique has an advantage over SPIN. In fact, the methods of Goubault et al. may not be optimally efficient in reducing the state space, and a modification that can further reduce the state space explosion problem is considered here. Experiments have shown that Goubault's inductive method produces a number of components that is approximately quadratic in the number of semaphores for the two-process progress graphs being considered; however, they have also demonstrated that the modified approach lowers the exponent to approximately one.

This modification of Goubault's method and its correctness are described later and in a technical report [19]. As geometric semantics makes its way into general model checking software (especially for deadlock detection), the reduced asymptotic complexity of this method of partitioning the state space will make an improvement in efficiency possible. If these algorithms can be exploited for general use in model checking LTL formulas over traces [20], then a significant benefit to model checking methods can be expected from this improvement because there are exponentially many paths to be checked. Addressing the state space explosion problem for deadlock detection is but a first step toward efficient general model checking for LTL or CTL formulas, but the generalization may not be so difficult, especially for one familiar with the algorithmics of the latest model checking software.

3. GEOMETRIC SEMANTICS

3.1. DEFINITIONS

Definition 1. An *open rectangle* is a product of two open intervals of real numbers (one for each process of a two-process concurrent program).

Definition 2. A *forward-closed rectangle* is a product of two half-open, right-closed intervals of real numbers, and a *backwards-closed rectangle* is a product of two half-open, left-closed intervals of real numbers.

Definition 3. A *closed rectangle* is a product of two closed intervals of real numbers. Therefore, a *point* is a degenerate case of a closed rectangle.

Definition 4. A *two-dimensional progress graph with (only) significant points* is a closed rectangle of real numbers along with a (possibly empty) subset that consists of distinguished isolated points (called significant) in the interior of the rectangle.

Definition 5. A *two-dimensional progress graph with (only) forbidden points* is a closed rectangle of real numbers minus a (possibly empty) subset that consists of significant points.

This work is concerned with how paths pass through or around the significant points; however, for forbidden points, it necessarily considers only how paths pass around them.

Definition 6. A *two-dimensional progress graph with forbidden rectangles* is a closed rectangle minus the union of a subset of (possibly intersecting, possibly degenerate) forward-closed rectangles that have their interior and boundary in the interior of the progress graph and (by definition) have edges oriented parallel to the coordinate axes. See Figure 2.4.

Definition 7. The *codimension* of a linear algebraic set in Euclidean space is the minimum number of linear equations needed to define it. For example, the codimension of a point in a two-dimensional progress graph is equal to two (one for each coordinate).

Definition 8. A *significant column* is an inverse image (of the natural projection function) of a significant point in a two-dimensional progress graph equal to an initial two-dimensional face of an N-dimensional hypercube (i.e., containing the origin and therefore not the terminal point of the hypercube).

Definition 9. A *forbidden column of codimension two* is the natural analogue of a significant column, where the point is forbidden instead of significant.

Definition 10. More generally, a *forbidden column* is an inverse image (of the natural projection function) of a forbidden rectangle in a two-dimensional progress graph equal to an initial two-dimensional face of an N-dimensional hypercube.

Definition 11. An *N-dimensional progress graph with significant columns* is an N-ary product of closed intervals of real numbers along with a finite number of significant distinguished subsets, each a significant column. A significant column may be described with a notation for a point, where "don't care," or '*' values, are placed in the free dimensions. For example, in a three-dimensional graph, the significant set $(0.2, *, 0.5)$ allows x_1 to vary but fixes x_0 equal to 0.2 and x_2 equal to 0.5.

Definition 12. An *N-dimensional progress graph with forbidden columns (possibly of codimension two)* is an N-dimensional progress graph with significant columns minus a set of forbidden columns (possibly of codimension two).

Definition 13. For a coordinate x_i that is fixed for a forbidden or significant point F (where x_j is the other fixed coordinate), the *extension* $E(F, j)$ is the hyperplane through F that has only x_i fixed.

Definition 14. The *topological space of the flow graph of a process* is a topological space obtained by connecting the nodes and edges of the flow graph of a process. It has a finite number of points where nodes of the flow graph have an out- or in-degree greater than one, where branch segments may branch apart and join together. Moreover, loops are permitted unless otherwise indicated.

Definition 15. A *progress graph without branching* is a product of the topological spaces of the flow graphs of individual processes that do not have branches.

Definition 16. A *progress graph without looping* is a product of the flow graphs of processes that do not contain loops.

Definition 17. A *locally partially-ordered topological space* is a topological space along with a covering by compatible partially-ordered open subsets.

Definition 18. The past/future of a point x in a progress graph is the subset of points of the progress graph that are less/greater than x in the partial-order.

Definition 19. A *dipath* in a progress graph is a continuous function from a closed interval (most often [0,1]) into the progress graph that is nondecreasing in each coordinate. This work is concerned primarily with dipaths that map 0 to the minimal point, in two dimensions, the lower left corner) of the progress graph. The component dipath function will be written $p(t) = (p_0(t), p_1(t))$.

Because a dipath may pass directly through a significant point or corner of a significant rectangle, the interpretation of this situation in terms of the execution sequence of the statements of the processes must be universally accepted. Therefore, an additional definition will be helpful.

Definition 20. An *admissible dipath* is one that has the following property, which clearly guarantees the unambiguity of the sequence of transitions taken as it passes by individual points of the progress graph: it must not pass diagonally through any significant point when the dipath is projected into each two-dimensional subspace of the progress graph (spanned by two of the axes).

One can unambiguously deform an inadmissible dipath into an admissible one by choosing a distinguished slope (zero, for example) to which to compare the upper-right Dini derivative [21] of the dipath as it passes diagonally through a corner. If the derivative is greater than the chosen slope, then the dipath must be deformed locally into one that passes through the corner vertically instead of diagonally or horizontally. If, on the other hand, the derivative is zero, then no deformation is necessary.

To simplify the proofs, the remainder of this dissertation assumes that the term dipath refers to admissible dipaths.

Definition 21. An *extension of a dipath* p is a dipath $q \neq p$ for which p is a restriction of q (so the domain of q must properly contain the domain of p).

Definition 22. A *maximal dipath* in a progress graph is a dipath that cannot be extended (backwards or forward) in the progress graph.

Definition 23. A *valid dipath* in a progress graph is a dipath that begins at a reachable point, meaning that there exists a dipath that begins at the minimal point (lower left corner) of the progress graph and ends at the reachable point.

Definition 24. A *successful dipath* in a progress graph is a dipath that ends at the maximal point (upper right corner) of the progress graph.

Definition 25. A *rectangular dipath* in an N-dimensional progress graph is a dipath that linearly changes only one of the coordinates x_0 through x_{N-1} at any given time, meaning that it is piecewise-linear and those linear segments are parallel to the axes.

Definition 26. A *dihomotopy* between dipaths f and g is a continuous function h from the unit square to the progress graph; h restricts to f on the bottom edge, restricts to g on the top edge, and takes the left and right edges to the initial and final points (respectively) of f and g, which must coincide at their endpoints. In addition, each horizontal cross-section between f and g must be a dipath. If a dihomotopy exists between f and g, then they are said to be *dihomotopically equivalent*. See Figure 2.2 for an illustration of dihomotopically equivalent paths.

Definition 27. Let p be a maximal dipath in a progress graph, and let S be a set of significant points of the progress graph. Then the *type* of p relative to S describes how the dipath passes around or through the points of S. For example, a dipath that goes to the left of and then above a particular point of S or vertically through the point indicates more progress relative to the significant point by the process that corresponds to the vertical axis, (referred to here as the second axis). Given such a dipath, for that point, the type data are $(1, 0)$. The 1 signifies that the dipath went above (not below) the point, and the 0 signifies that it went to the left (not to the right).

In an N-dimensional progress graph G with forbidden columns of codimension two, there are $\binom{N}{2}$ possible orientations for the columns, based on the fixed coordinates. Because distinct forbidden columns of codimension two with the same orientation do not intersect, the maximum number of forbidden columns that may intersect at a given point is $\binom{N}{2}$, and forbidden columns of the same orientation are isolated. Because all extensions are of codimension one and extend from end to end in $N-1$ coordinates, each divides G into two regions of points not on the extension (based on how the i-coordinate compares with x_i). This observation provides the basis for the definition of a higher-dimensional type for a particular dipath.

Let p be a maximal dipath in an N-dimensional progress graph with codimension-two significant columns, and let S be an ordered set of significant sets of the progress graph. Then the type of p relative to S describes how the dipath passes around or through the significant sets of S. The type data for p and an element F of S are in the form of an N-tuple, where the fixed i and j coordinates have either a 0 or a 1 in their locations, and the other coordinates (for which F is allowed to extend) are assigned '*' or "don't care" values. For example, a dipath p in a three-dimensional progress graph with only one significant column (a line segment, in this case) specified by (0.5,*,0.25) could have a type described by (0,*,1) if p crosses the plane $x_0 = 0.5$ before it crosses the plane $x_2 = 0.25$, or described by (1,*,0) otherwise.

Definition 28. A type is *admissible* if there is a dipath in a progress graph that has the given type. Otherwise, it is called *inadmissible*. This definition excludes types that cannot correspond to dipaths because of the requirement that dipaths progress forward in each coordinate.

Definition 29. The *digraph of a rectangularly partitioned progress graph* is one obtained by considering the rectangular components of the partition to be nodes and the codimension-one boundaries between them to be directed edges.

Definition 30. A *path in a digraph* is a path in the usual sense. Maximality and successfulness are defined as in the case of dipaths.

Definition 31. The *type of a path* p in a digraph (obtained from a rectangularly partitioned progress graph with significant or forbidden columns of codimension two) is the type of any maximal dipath that maps to p.

Figure 3.1 shows that the digraph of a partitioned progress graph may contain a path that has an inadmissible type – no dipath can go through the same components in the same sequence as that path does, because dipaths must not go backwards in any coordinate direction.

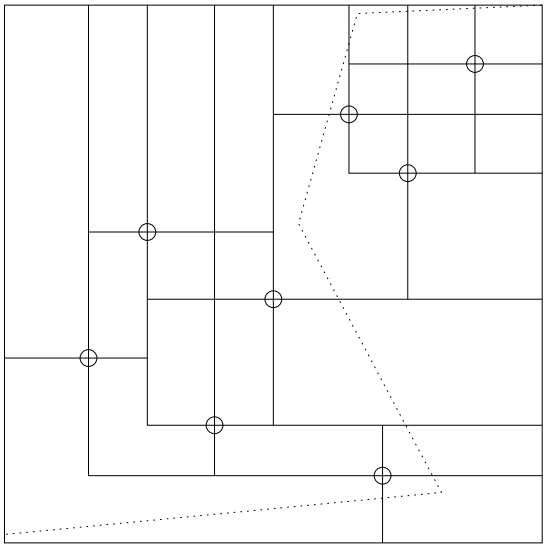

Figure 3.1. Small circles represent forbidden points. The path from the lower left corner to the upper right corner has an inadmissible type.

Definition 32. A *maximal component in a partitioned progress graph* is a component of the partition that corresponds to a maximal node in the associated digraph, meaning that there are no directed edges coming out of that node.

3.2. PROPERTIES

3.2.1. Two-Dimensional Case

Lemma 33. *Deadlock is impossible in a progress graph with forbidden points.*

Proof of Lemma 33. Topological reasons pertaining to compact sets imply that dipaths may come arbitrarily close to a forbidden point; however, a particular dipath always maintains some minimum positive distance from each forbidden point. Once this minimum distance is known, the dipath can be extended by that distance in the direction of one axis, thereby bypassing the point. Finally, because this work assumes that the forbidden points are isolated (and thus that there are a finite number of them), these extensions can be repeated until the maximal point of the (reversed) progress graph is reached. □

Corollary 34. *Deadlock is impossible in a progress graph with significant points.*

Proof of Corollary 34. The case of significant points is simple. Since it addresses admissible dipaths, significant points may be in the image of the dipath, but an exit (horizontal or vertical) from those points is easy to find, so deadlock does not occur as a result. □

Corollary 35. *In a progress graph with significant or forbidden points, every dipath is a valid dipath; equivalently, every point is reachable.*

Proof of Corollary 35. A point is reachable in a progress graph with significant or forbidden points if and only if one can reverse the direction of all axes and the point represents a safe state (not necessarily leading to deadlock). The latter condition is equivalent to the existence in the reversed progress graph of a dipath that connects the point in question to the maximal terminal point of the reversed progress graph. □

Corollary 36. *In a progress graph with significant or forbidden points, every maximal dipath is a successful dipath that begins at the initial corner of the progress graph.*

Lemma 37. *Let p be a dipath in a (two-dimensional) progress graph with forbidden points, and let (x, y) be a such a point of the progress graph. If p goes from the past of the point (x, y) to its future, then p goes below/above (x, y) if and only if it goes to the right/left of (x, y).*

Proof of Lemma 37. If p goes above a forbidden point (x, y), then it goes through a point (x, y_0) with $y_0 > y$. Let $t_0 \in (0, 1)$ be such that $p(t_0) = (x, y_0)$. Because p is a

dipath, it is continuous; therefore, its projection $p_1(t)$ is also continuous. Because $p_1(0)$ is equal to zero and $p_1(t_0)$ is equal to y_0, by the intermediate value theorem there exists a $t_1 \in (0, t_0)$ such that $p_1(t_1)$ is equal to y. That is, there is a $t_1 \in (0, t_0)$ such that $p(t_1)$ is equal to (x_1, y) for some x_1. Now, p is a dipath; therefore, x_1 is less than or equal to x because t_1 is less than t_0. However, x_1 is not equal to x because (x, y) is a forbidden point, meaning that p goes to the left of (x, y). The proof of the converse is trivial. □

Corollary 38. *Let p be a dipath in a two-dimensional progress graph with significant points, and let (x, y) be such a point in the progress graph. If p goes from the past of the point (x, y) to its future, then p goes below/above (x, y) if and only if it goes to the right/left of (x, y). We use the terms below, above, right, and left as we would when assigning the type data to the dipath.*

Proof of Corollary 38. If p goes through the point (x, y) horizontally, then it cannot have gone above (x, y) previously. The case of p going through (x, y) vertically is similar. □

One can conclude that the only possible type data for a point of S are $(0, 1)$ or $(1, 0)$ due to the nondecreasing properties of the dipath, which are essential for the proof of Lemma 37. This definition may seem redundant, but it has been written for ease of use when generalizing to codimension-two significant or forbidden columns in higher-dimensional progress graphs. In the case of a dipath passing directly through a significant point, the type data is assigned according to the trajectory by which the dipath exits from the point in question.

3.2.2. Higher-Dimensional Case All the concepts of the previous subsection can be generalized to higher dimensions. Significant or forbidden points are replaced by significant or forbidden columns of codimension two. The analogue of a forbidden rectangle is a forbidden column. The only interesting feature of the generalization is that the forbidden columns can be oriented differently (with respect to each other).

Lemma 39. *Restriction of an N-dimensional progress graph with its significant or forbidden columns, $N \geq 3$, to the set of points where a certain coordinate is maximal or minimal forms an $(N-1)$-dimensional progress graph. Similar subspaces of this graph can form*

progress graphs of dimension two to $N-1$. The union of these subspaces describes the entire boundary of the N-dimensional progress graph. Projections of a dipath in the N-dimensional progress graph to any of these lower-dimensional graphs are still dipaths, and similarly, dipaths in any of the lower-dimensional graphs are dipaths in the N-dimensional graph.

Proof of Lemma 39. By induction on dimension, only the $(N-1)$-dimensional subspaces need be treated. When one coordinate is fixed to obtain the lower-dimensional progress graph, some forbidden and significant columns no longer need be considered because they do not intersect it. Consider the ones that do intersect the lower-dimensional progress graph. They are bounded in two directions by intervals that are not equal to the entire extent of the progress graph along those axes. For a particular column, if the coordinate that was fixed *is not* associated with one of those two directions, then its restriction remains a forbidden or significant column in the lower-dimensional progress graph. On the other hand, if one assumes that it *is* associated with one of those two directions, then it cannot intersect the lower-dimensional progress graph at all, because the fixed coordinate was assumed to be maximal or minimal. □

Theorem 40. *In an N-dimensional progress graph with forbidden columns of codimension two, every dipath is valid.*

Proof of Theorem 40. Let x be a point in a progress graph G fitting the conditions of the theorem, and assume that x is not in a forbidden column. Clearly, there is some two-dimensional progress graph H on the boundary of G such that the image of x under projection to H is not forbidden. To show that x is reachable in G, a dipath p from the origin to the projection of x in H will be constructed; then, p will be broken into subpaths that can be translated from H to G in directions orthogonal to the plane that contains H, and these subpaths will be connected by simple line segments to complete the proof of the theorem. Let i be the least index of all of the axes that are orthogonal to H. There is a progress graph J contained in G that extends H in the direction of the i axis, and it has a finite number of new forbidden columns in it. If the inverse image of p under projection from J to H intersects m of those columns, then it is possible to break the dipath into

$m+1$ subpaths that may be translated and reconnected as needed to go around the new forbidden columns. This procedure can be repeated for each new dimension that is taken into consideration until a dipath in G from the origin to x is obtained. □

Corollary 41. *In an N-dimensional progress graph with codimension-two significant columns, every dipath is valid.*

Corollary 42. *In an N-dimensional progress graph with codimension-two significant or forbidden columns, every maximal dipath is a successful dipath that begins at the initial corner of the progress graph.*

Lemma 43. *Let p be a maximal dipath in an N-dimensional progress graph with significant or forbidden columns of codimension two, and let F be a codimension-two significant or forbidden column of the progress graph. Then the type data of p relative to F contain exactly one 0 and exactly one 1.*

Proof of Lemma 43. One can simply project p onto the appropriate two-dimensional face, which is a progress graph with significant or forbidden points. □

Lemma 44. *Let f be a maximal dipath in a progress graph with codimension-two significant or forbidden columns. Then f can be approximated by a rectangular dipath j such that f and j are dihomotopically equivalent.*

Proof of Lemma 44. Leaving the case of significant points to the reader, the lemma is proved here for forbidden points only. To approximate f, let d be the minimum of the Manhattan distance from f to any boundary or forbidden point. A rectangular dipath j so close to f that it must be dihomotopically equivalent to f will be defined. Partition [0,1] by $0 = a_0 \leq a_1 \leq \ldots \leq a_m = 1$ such that for all i, the distance from $f(a_{i-1})$ to $f(a_i)$ is less than d. Next, for each interval, let r_i be equal to $(a_i - a_{i-1})/N$, and define

$$j(a_{i-1}) = f(a_{i-1}),$$

$$j(a_{i-1} + r_i) = (f_0(a_i), f_1(a_{i-1}), f_2(a_{i-1}), \ldots, f_{N-1}(a_{i-1})),$$

$$j(a_{i-1} + 2r_i) = (f_0(a_i), f_1(a_i), f_2(a_{i-1}), \ldots, f_{N-1}(a_{i-1})),$$

$$\ldots,$$

$$j(a_i) = f(a_i).$$

These points can be connected linearly, and a weighted average dihomotopy (with t as a weight parameter) can be constructed between f and j. □

Theorem 45. *If f and g are maximal rectangular dipaths in a progress graph with significant or forbidden codimension-two columns of the same type relative to all the significant or forbidden columns, then they are dihomotopically equivalent (i.e., there is a dihomotopy between f and g).*

Proof of Theorem 45. The case of codimension-two forbidden columns is presented here. Because rectangular dipaths involve changing exactly one coordinate at a time, they are made up of line segments parallel to the coordinate axes. There are two cases to consider, but both parts of the proof use a similar strategy. That is, a sequence of dipaths will be defined that are each dihomotopically equivalent to g, and each dipath in the sequence will be dihomotopically equivalent to f on a longer initial part of its image. Begin the sequence with g_0 equal to g. Let r_k be the minimum distance of all of the line segments that make up f and g_k. This is helpful because if f or g_k extends in one direction in a maximal interval $[t_i, t_f]$, then the distance traveled in that direction by the dipath is at least r_k.

Case 1. Suppose that for two rectangular dipaths f and g_k, $f(t_k)$ is equal to $g_k(t_k)$, which is equal to P_k, and f and g_k extend from P_k in the same direction to some maximal point P_{k+1}. Reparameterize f and g_k by defining a $t_{k+1} > t_k$ such that $f(t_{k+1})$ is equal to $g_k(t_{k+1})$, which is equal to P_{k+1}, and such that for any t in $[t_k, t_{k+1}]$, $f(t)$ is equal to $g_k(t)$. In this case, f and g_k are clearly dihomotopically equivalent on this line segment because they coincide. For the sake of generality, define g_{k+1} as equal to g_k, meaning g_k and g_{k+1} are dihomotopically equivalent, and f and g_{k+1} are dihomotopically equivalent from P_k to P_{k+1}, leading to a Manhattan distance decrease from the terminal corner of at least r in traveling from P_k to P_{k+1}. Further, $f(t_{k+1})$ is equal to $g_{k+1}(t_{k+1})$, which is equal to P_{k+1}.

Case 2. Suppose that for two rectangular dipaths f and g_k, $f(t_k)$ is equal to $g_k(t_k)$, which is equal to P_k, and f and g_k extend in different directions from P_k. Specifically,

suppose that f extends in the direction of the i axis by some $a \geq r_k$ and g_k extends in the direction of the j axis. Clearly, then, the i and j coordinates of P_k are not maximal; therefore, a later section of g_k must include an increase in the i coordinate to some maximal point of a line segment by some distance $b \geq r_k$. Define c as equal to $\min\{a, b\}$, and define P_{k+1} as the point that is c units greater than P_k in the i direction (a point on f). Let g_{k+1} be a rectangular dipath that agrees with g_k up to P_k, but travels from P_k to P_{k+1}, and then is simply a translation of g_k by c units in the i direction, until coinciding with g_k once more. The new dipath g_{k+1} is clearly dihomotopically equivalent to f on the line segment from P_k to P_{k+1} (as they coincide); therefore, it is now necessary to show that g_k is dihomotopically equivalent to g_{k+1}.

Because g_k and g_{k+1} only differ from P_k up to their later point of intersection, this is the only section of concern. The curves traced by g_k and g_{k+1} form rectangles with directions parallel to coordinate axes (one of which is always the i-axis). Therefore, parameterize g_k and g_{k+1} so that they simultaneously cross corresponding vertices that are intersections of distinct rectangles. To do this, hold g_k constant until g_{k+1} reaches P_{k+1} at some $t_{k+1} > t_k$, parameterize the parallel pieces of g_k and g_{k+1} by arc length, then hold g_{k+1} constant until g_k reaches the new intersection point. Also, parameterize f so that $f(t_{k+1})$ is equal to $g_{k+1}(t_{k+1})$, which is equal to P_{k+1}, and for any t in $[t_k, t_{k+1}]$, $f(t)$ equals $g_{k+1}(t)$. This procedure ensures that for any value of t in $[0, 1]$, a line segment from $g_k(t)$ to $g_{k+1}(t)$ is either on the curve traced by g_k or g_{k+1} (often degenerating to a point of intersection) or in one of the rectangles formed. As such, the linear combination dihomotopy $h_s(t) = h(s, t) = s \cdot g_k(t) + (1-s)g_{k+1}(t)$, will be contained either on the curves or in the rectangles, and because this is nondecreasing in each coordinate as shown by previous arguments, it is now only necessary to show that the rectangles do not intersect forbidden columns.

Because g_k and g_{k+1} differ only in the i coordinate whenever they do not coincide, the rectangles cannot intersect a codimension-two forbidden column that is allowed to extend in the i coordinate, for this would imply that g_k intersects that forbidden column. Therefore, the only other case to consider is a forbidden column that is fixed in the i coordinate. If m is the other fixed coordinate for a forbidden column F that intersects

one of the rectangles, then $E(F, i)$ extends in the i direction. Because g_k is the minimum for the rectangles in the i coordinate and because it cannot intersect the forbidden column itself, it intersects $E(F, i)$ in the lower i coordinate region. However, f extends in the i coordinate from P_k to at least the level of g_{k+1} (i.e., the level of P_{k+1}), the maximum for the rectangles in the i coordinate, and the fixed m coordinate of F must be greater than that of P_k (because otherwise f would intersect F). Therefore, f intersects $E(F, m)$ in the lower m coordinate region, which is the opposite of the type data for g_k. Thus, if f and g_k have the same type, the rectangles do not intersect forbidden columns; therefore, g_k and g_{k+1} are dihomotopically equivalent, and f and g_{k+1} are dihomotopically equivalent at least up to P_{k+1}. P_{k+1} is closer to the terminal corner of the progress graph than P_k by at least c units. Further, $f(t_{k+1})$ is equal to $g_{k+1}(t_{k+1})$, which is equal to P_{k+1}.

For the rectangular dipaths f and $g_0 = g$, define P_0 as the initial corner and t_0 as equal to zero, so that $f(t_0)$ is equal to $g_0(t_0)$, which is equal to P_0. From that point, based on which of the two cases f and g_0 follow after P_0, one can prove that g_0 is dihomotopically equivalent to g_1 (another maximal dipath) on the whole progress graph, and that f and g_1 are dihomotopically equivalent on a line segment from P_0 to a point P_1 that is at least r_0 units closer in Manhattan distance to the terminal corner than P_0. One can further prove that there is a $t_1 > t_0$ such that $f(t_1)$ is equal to $g_1(t_1)$, which is equal to P_1. If P_1 is not the terminal corner, then continue this cycle for t_2, P_2, g_2, t_3, P_3, g_3, and so on, as each case concludes with dipaths that coincide in order to continue the iteration.

Consider the sequence (P_i). For the sake of argument, suppose that it does not converge to the terminal point of the progress graph. In that case, it converges to another point that is in the interior of the progress graph. Let that point be called P_a. From P_a, the procedure can begin again, continuing up to the next point of convergence, P_b. In fact, there is always a way to continue the procedure to reduce the distance remaining to the terminal point of the progress graph. Even if the sequence $P_a, P_b, \ldots$ converges before the terminal point, there is a way to continue again past that point of convergence. Therefore, there can be no supremum (other than the terminal point) of the set of points through which the desired dihomotopy can be constructed. The transitive property of dihomotopic equivalence assures us that f and g are equivalent. □

Corollary 46. *For any dipaths f and g (not necessarily rectangular) that satisfy the conditions of Theorem 45, f and g are dihomotopically equivalent.*

Proof of Corollary 46. Based on Lemma 44, there are rectangular approximations to the dipaths f and g, f' and g', such that f and f' are dihomotopically equivalent (and likewise for g and g'). Note that f' and g' are maximal. Further, f and f' have the same type, as do g and g'. Therefore, Theorem 45 applies to f' and g', and dihomotopy equivalence is transitive, so the conclusion follows. □

4. DIHOMOTOPIC REDUCTION

4.1. COMPONENT GRAPHS

This section examines the use of geometry to detect deadlock for various types of model programs. The discussion begins with simple, restricted models and then moves on to include more realistic features.

4.1.1. Inductive Decomposition The first application of dihomotopic state space reduction was announced by Goubault and Raussen [9]. They used a partitioning method based on the minimal and maximal corners of forbidden rectangles to aggregate states into components, between which there were transitions determined by shared boundaries. The present work first altered their method simply by changing the partitioning method, thus revealing that the asymptotic complexity of the state space can be reduced relative to the number of semaphore operations executed by a typical process.

This section first addresses the decomposition of a very restricted model program having processes with semaphore operation as statements. There follows an explanation of how the method can be generalized to accommodate programs that include processes which not only lock and unlock semaphores, but also read from global variables and write to them as well. Computational operations on other variables are permitted as well. The final subsection describes the modifications to our method that are intended to address nondeterministic guarded branch selection and looping.

Concurrent programs possess a natural geometric structure that contains enough information to detect unsafe and unreachable regions of their state spaces, and Goubault, Raussen, Fajstrup, and others have shown that this structure is amenable to exploitation for efficient algorithmic deadlock detection. They proposed that a state space should be partitioned both along the boundary of the *past states* of the minimal corner of each forbidden rectangle and along the boundary of the *future states* of the maximal corner of each forbidden rectangle. They were able to show that this partitioning method is fine enough to detect deadlock and at the same time is coarse enough to permit aggregation of states into a reduced state space for exploration.

Goubault's dihomotopic state space reduction works particularly well because the statements of the processes are simple, consisting solely of semaphore locks and unlocks. Even so, it is a revolutionary concept and deserves further attention because it points the way to a more sophisticated approach along the same lines, an approach that has the potential to improve significantly upon the standard technique of partial-order reduction. The author and his collaborators have treated a generalized version of the problem that was solved by Goubault's method.

Inductive partitioning has been shown to reduce the state space for the dining philosophers problem. An alternative partitioning method can improve the performance when a progress graph contains a sufficient number of forbidden regions. The methods for both types of decomposition (for concurrent programs of two processes) are described next, along with their means of preserving the properties of the state transition system that are important for deadlock detection. To generalize to more processes, one considers pairs of processes and then overlays the boundaries that ensue in a higher-dimensional progress graph.

Goubault's method is to partition a progress graph into components inductively by overlaying the boundaries they use to partition a base case of only one semaphore (resulting in only one forbidden square somewhere in the middle of the progress graph). This approach is very interesting from a purely mathematical perspective, but perhaps not optimal for reducing the complexity of the state space. Experiments described by Goubault and Haucourt [15] demonstrate that, in some cases, an advantage can be obtained over the technique of partial-order reduction used by the popular model checking software SPIN. Namely, the dining philosopher problem can be addressed for varying numbers of philosophers.

There is a correspondence between forbidden rectangles and semaphores: the minimal (lower left) corner of a forbidden rectangle corresponds to locking and the maximal (upper right) corner corresponds to unlocking a semaphore by each process of the progress graph. According to the algorithm developed by Goubault, the rectangles are addressed one at a time, drawing a horizontal ray from the minimal point back to the left boundary of the progress graph, and drawing a vertical ray down to the bottom boundary of the

progress graph. Similarly, horizontal and vertical rays are drawn from the maximal point of the forbidden rectangle to the right and top boundaries of the progress graph. These steps are repeated for each rectangle, as shown in the three examples in Figure 4.1.

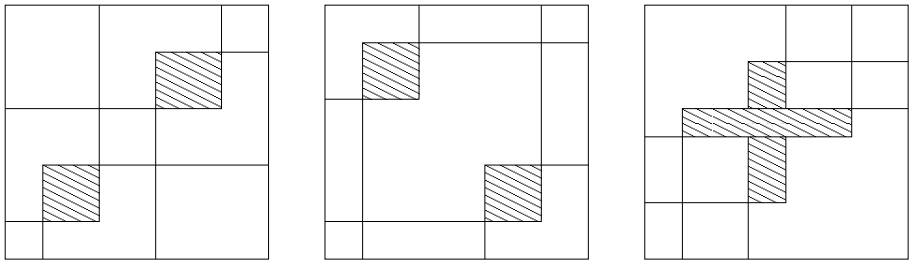

Figure 4.1. Inductive partitioning for two semaphores; thus, m=2. In the first two progress graphs, there are 9 regions; in the third, there are 10 regions, but one is unreachable. In general, approximately $(m+1)^2$ regions are expected.

This procedure divides the progress graph into regions, which are called *components*. A digraph is formed with these components as nodes and vertices, and the boundaries (line segments) between them as directed edges (where crossing a boundary represents a transition). Certain squares in the corresponding digraph are commutative in the sense that traversing one or the other path from the initial point of the square to the terminal point is not distinguishable from the point of view of dihomotopy. The purpose of the partitioning is to define equivalence classes in such a way that the sequence of regions through which a path travels determines a unique Mazurkiewicz trace, and these traces correspond to dihomotopy equivalence classes. Goubault and Raussen [9] describe a possible optimization, but the procedure just described expresses the essential idea of the inductive algorithm. The optimization eliminates the extension of boundary partitions to the boundary of the progress graph when they first intersect another forbidden region.

For dimensions higher than two, an earlier paper by Goubault and Haucourt [15] describes the algorithm presented by Goubault et al. for forbidden cuboids using "precubical sets." It offers a straightforward modification to handle simple codimension-two forbidden columns as in earlier sections.

4.1.2. Cross-Sectional Recursive Decomposition

4.1.2.1. Box-Shaped Shells The impetus for this dissertation project came from an Internet search using terms such as *concurrency* and *geometry*. In an effort to increase the efficiency of the algorithm, differently shaped components were used in the decomposition of progress graphs. Little progress was achieved, however, until the easier problem was considered of representing dihomotopy classes of dipaths in progress graphs that had no forbidden rectangles, but forbidden points instead. In addition, the optimization that Goubault and Raussen uses shorter boundary lines. These considerations led to the discovery that a partition similar to the inductive one (but with boundary lines of different length) could be applied to the problem of deadlock detection.

When forbidden points are the only obstacles to dihomotopic deformations of dipaths, it is much easier to see that fewer boundaries are needed to represent all traces. In fact, the inductive partition is too fine, and a procedure is required to remove some of the unnecessary boundary separations between rectangular components. A human being can easily determine various ways to merge such components; however, a reasonable algorithm is needed to do so. Initially, this work addressed the problem by using box-shaped shells for partitioning progress graphs. This kind of decomposition is based on the two aspects of correct deadlock detection – faithful capture of both maximality and reachability for each potential point of deadlock. During trial and error, the author observed that the only boundaries needed near a forbidden point are those in the shells very close to any particular forbidden point of interest; therefore, the boundaries may be taken to be quite short indeed, as illustrated in Figure 4.2.

4.1.2.2. L-Shaped Cross-Sections Unfortunately, in terms of computer science, there is no easy interpretation of formation of box-shaped shells for partitioning a progress graph. The next technique applied, though slightly simpler, was not much more intuitively appealing. After the procedure was recognized as being somewhat recursive, the box-shaped shells were abandoned in favor of L-shaped shells (i.e., the lower left half of a box-shaped shell). The theory developed for that recursive partitioning method has been described in a technical report [19] and a conference paper [22]. Later, some important modifications

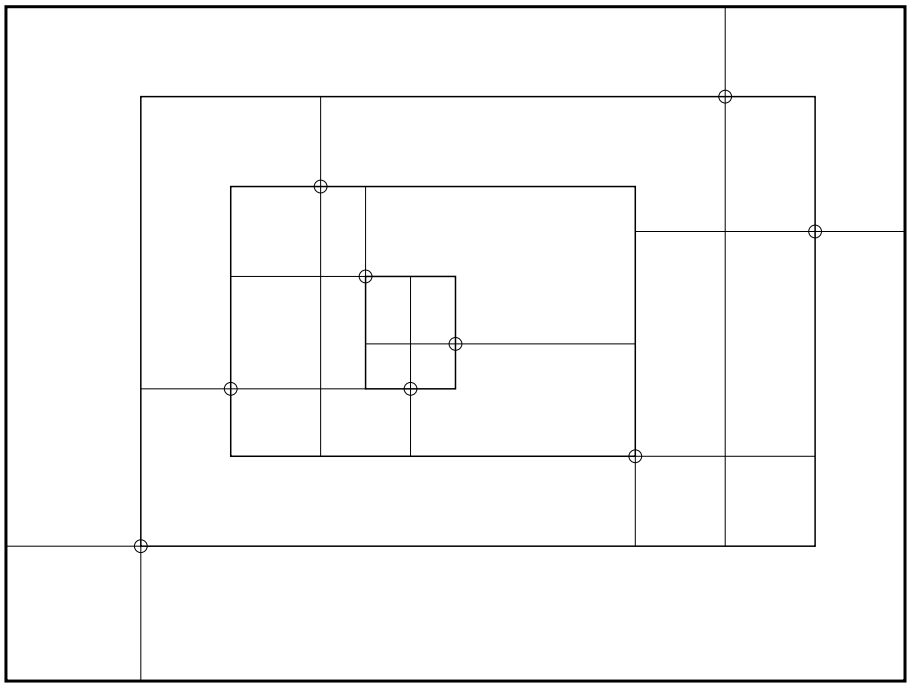

Figure 4.2. Box-shaped shells were used in the first attempt at cross-sectional decomposition of a progress graph with forbidden points.

to the method were described in another conference paper [23]. For completeness, we include the algorithm and some of those results here.

4.1.2.3. Recursive Reduction Algorithm The algorithm for forbidden points only is discussed first. Let V be the set of forbidden points. Next, let B_0 be the entire progress graph, and let B_1 be the smallest rectangle inside B_0 which contains all of the points of $V_0 = V$ and contains the upper right corner of the progress graph. The left and bottom sides of B_1 will contain at least one point of V_0, either in the interior or at the corner. Let V_0' be the set of points of V_0 which are on the left or bottom sides of B_1, and let $V_1 = V_0 - V_0'$. Repeat this procedure to define B_l, V_l, and V_l' for each $l \geq 0$ until V is exhausted. Let the closed sets $N_0 = closure(B_0 - B_1), N_1 = closure(B_1 - B_2), \ldots$, and call $\{N_0, N_1, \ldots\}$ the initial partition of the progress graph.

Next, partition each N_l according to the positions of the points of $V'_{(l-1)}$ and V_l'. For example, see Figure 4.3, which represents a two-dimensional case. The basic idea is that we draw L-shaped regions and then partition them with line segments.

Since each point of V_l' is on at least one face of B_{l+1}, one can define the direction(s) normal to the faces that contain a given point of V_l'. For each such direction, extend

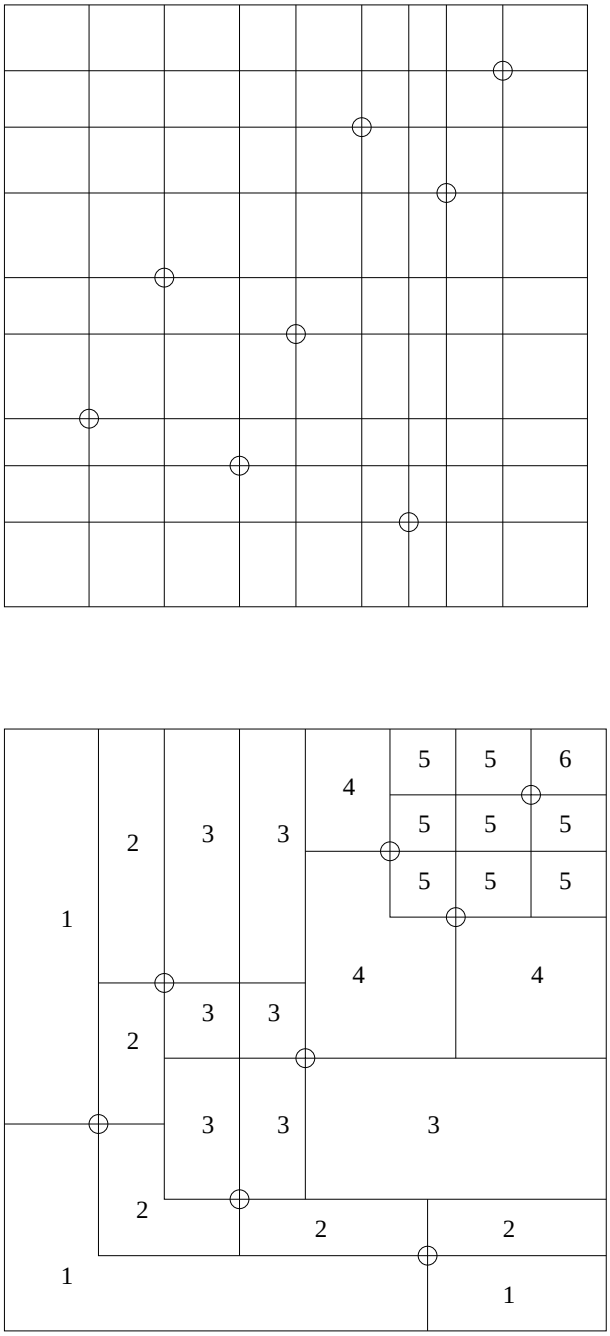

Figure 4.3. Small circles represent forbidden points. Numbers correspond to L-shaped regions in the initial partition. The inductively partitioned progress graph is at the top, and the recursively partitioned one is below.

line segments to the left or down (outward) from the inner boundary of N_l to the outer boundary of N_l (remember that N_l is the complement of a L-shaped region). Also extend line segments to the right or up (inward) through N_l from the points of $V'_{(l-1)}$. This procedure also produces a transition digraph.

Now consider the case in which forbidden rectangles are permitted. Collect all of the corners of all of the forbidden rectangles C_j into a set $V = \cup_{j=1}^{m} V_j$. then follow the procedure for forbidden points, except that a slight modification is required because some of the L-shaped regions obtained may be crossed by forbidden rectangles. To accomodate this possibility, simply add new vertices to V wherever a forbidden region crosses the boundary of an L-shaped region. The basic idea is still that we draw L-shaped regions and then partition them with line segments. See Figure 4.4, Figure 4.5, and Figure 4.6.

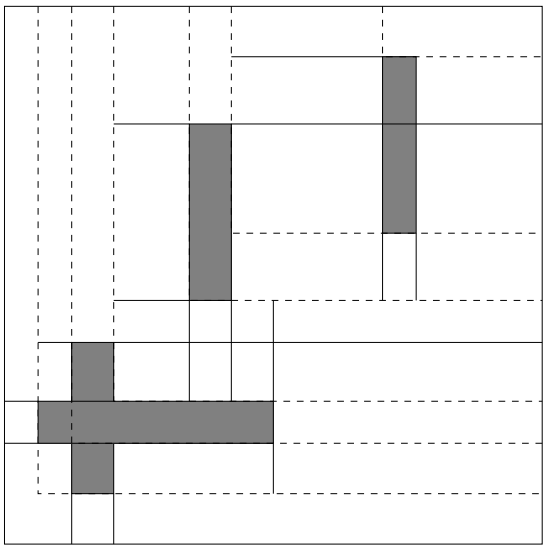

Figure 4.4. Shaded regions are forbidden, dashed lines make the initial partition, and solid lines complete the partition.

4.1.3. Complexity As far as complexity goes, the Recursive Reduction Algorithm has been found experimentally to produce approximately $O(m)$ regions, where m is the number of significant sets in a progress graph without branches or loops. This result may be due to the fact that each L-shaped boundary "consumes" at least two vertices, and the number of components in each L-shaped shell is bounded by one more than the number of forbidden sets in the progress graph that intersect it. It is unlikely that this upper bound will be attained very often, and it seems more reasonable to use an estimate of $O(k)$, where k is the maximum nested depth of a critical section. Therefore, we obtain an estimate of $O(km)$, or $O(m)$ if k itself is bounded by a constant.

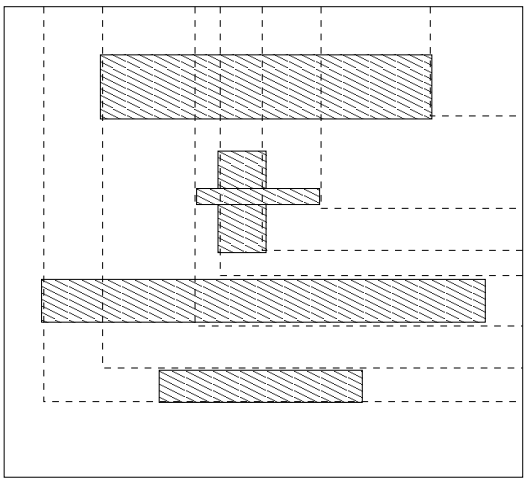

Figure 4.5. The long horizontal forbidden region at the top crosses through many L-shaped regions, showing the general need to add vertices to protect forbidden regions.

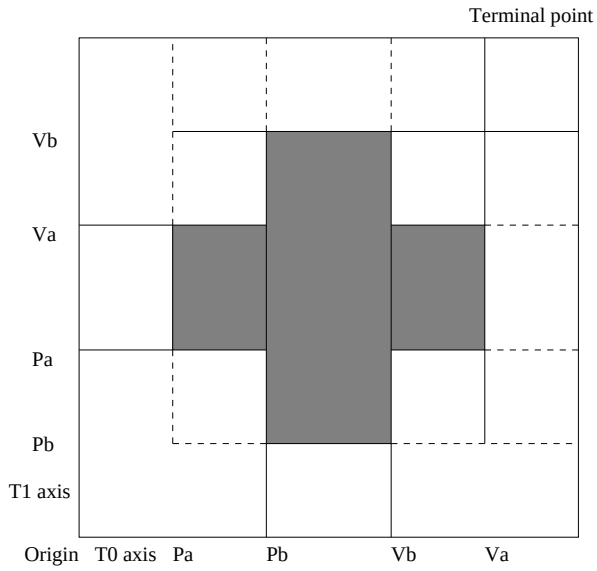

Figure 4.6. This is the recursively-partitioned Swiss flag example, with dashed lines for the initial partition.

For the unoptimized version of the inductive partitioning algorithm, applied to progress graphs without branches or loops, it is possible to estimate the number of components that will be produced, as shown in Figure 4.1. Each significant rectangle splits the progress graph vertically into a region to its left and a region to its right, and likewise horizontally into an upper and lower region. When this process is repeated m times, where

m is the number of significant sets (rectangles or points), there will be $m+1$ vertical strips produced as well as $m+1$ horizontal strips. Because the boundaries between the vertical strips overlay the boundaries between the horizontal strips, there will be a total of $\Theta(m^2)$ regions created in the progress graph.

4.2. STATE SEQUENCES

4.2.1. Reduced State Graphs When individual statements must be executed to determine changes in values of data variables that might impact deadlock (when, for example, a guard might be open or closed), state sequences rather than component sequences are necessary. State sequences, however, automatically prevent the search from exploring inadmissible maximal paths through the digraph obtained from a partitioned progress graph. Thus, there is no need for concern about inadmissible paths during the verification, because the search takes only forward state transitions. In fact, this observation brings up the issue of how the search actually does take place when individual states must be considered because there are computational steps inside some of the components themselves. The reduction of state space size by aggregating states has into geometric components has already been explained. The following discussion addresses the problem of programs including processes that use data variables and read and write statements enabling meaningful assignment statements and some basic interprocess communication.

The standard method of partial-order reduction, based on pruning unnecessary transitions, rather than considering aggregate states formed by equivalence classes in components, is applicable at this point. More explicitly, using a reduced set of transitions within the state space is justified because the computational steps specified by the statements must be performed during traversal of an execution path.

Following is a comparison of partial-order reduction and the proposed dihomotopic reduction. The latter uses a rectangular partition to select the smallest possible set of transitions between states to ensure that every reachable component is visited. Partial-order reduction also uses a small set of transitions, but the selection method is based mostly on noninterference between transitions in different processes. Here, the processes are used to determine the partition; the geometry is then allowed to take over so the content of the statements can be disregarded when choosing transitions for execution

paths. The actual procedure for basing a state search on a geometric decomposition is explained further in Section 5.

4.2.2. Progress Shell Decomposition

4.2.2.1. Initial Partition This section describes how the cross-sectional partitioning by progress shells is performed on two-dimensional progress graphs. First, the two constituent processes are analyzed to find the significant points and rectangles that will define the partition. For example, the points are identified in the progress graph where each process attempts to write to the same global variable, or alternatively where one process is attempting to read and the other is attempting to write. Another example is a case in which each of the two processes attempts to lock a semaphore. These are the points at which it matters which process executes first.

Once those important points in the progress graph are identified, they are projected onto each of the individual process axes to find the significant coordinates. Shells are then formed by matching the significant coordinate of the first process closest to the origin with the significant coordinate of the second process that is closest to the origin. By matching up the significant coordinates on each axis in sequence, a sequence of points called (middle) corners is obtained. In case of branching in a process, we employ an equalizing procedure to ensure that the partitioning continues in a well-defined manner after the branches join back together. Loops are treated as a special type of branching structure.

A shell has an inner (posterior) and outer (anterior) boundary. A boundary is the union of two line segments emanating from the corner, one vertically downward to the horizontal axis (representing the flow graph of the first process) and the other horizontally leftward to the vertical axis (representing the flow graph of the second process). The two-dimensional space between two consecutive boundaries is called a (progress) shell, and the formation of such shells is called the initial partition.

4.2.2.2. Secondary Partition Once the initial partition has been created, the individual shells are divided into rectangular components. First, the shell is separated into three rectangles consisting of the upper left, middle, and lower right regions. Next, the positions of the significant points and the corners of the significant rectangles are used to determine additional divisions. These locations are depicted in Figure 4.7.

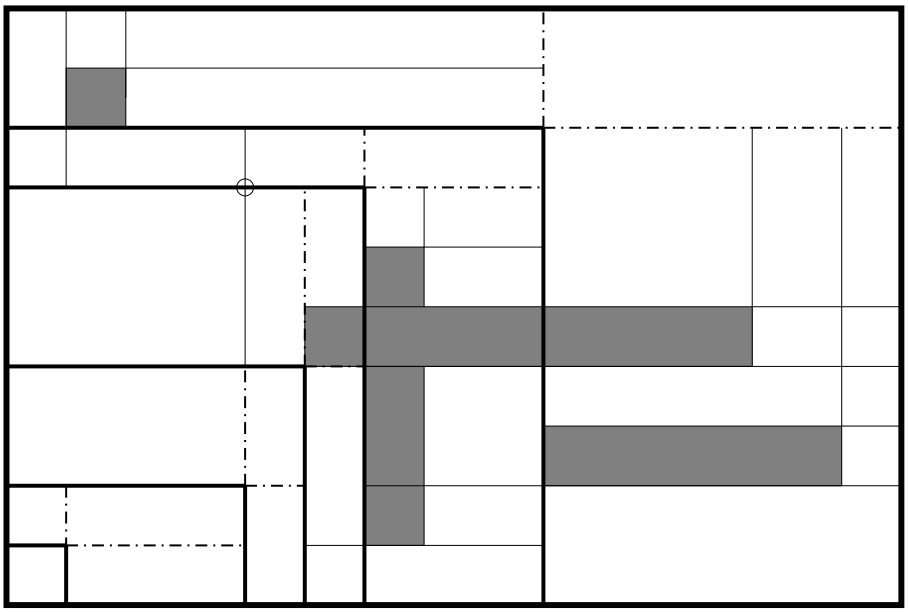

Figure 4.7. Here, we see a small typical example of a progress graph after the initial and secondary partitioning has been done. The small circle represents a signicant point, and the shaded rectangles are the significant rectangles.

Minimal corners of significant rectangles occur only on boundaries because the significant coordinates were used to determine those boundaries. The other three corners of a significant rectangle, however, often occur in the interior of a shell rather than on a boundary. The secondary partitioning ensures the correctness properties of the partition that enable us to detect deadlock and to represent program execution paths accurately. It also prevents odd-shaped components from being present in the final partition: all components should be rectangles for simplicity of the later search.

The left and lower separating line segments are extended from the minimal corner of each significant rectangle downward or leftward through the previous shell. The line segment is then extended forward (upward or rightward) past its diagonal corners to the next boundary. The line segments are also extended from the maximal corner forward (upward or rightward) to the next boundary as well. For the purposes of secondary partitioning, each significant point is considered to be the same as a degenerate significant rectangle (minimal and maximal corners coinciding).

4.2.2.3. Branches and Loops When there is a branching structure in one process of the pair being used to form the two-dimensional progress graph, the significant corners of each branch are identified separately. Because the other process may have some coordinates that are significant with respect to one branch and other coordinates that are significant with respect to the other branch, these two sets are combined. Shells are formed in the branched two-dimensional space of the progress graph according to the same procedure that was described for nonbranching progress graphs. The only real difference is in the secondary partition because a boundary is added at the beginning of the branching structure and at its end so that all rectangles are contained entirely in one branch or the other, or in common space before or after the branching structure itself.

To ensure compatibility between the shells as the branches join back together, *virtual* significant coordinates are inserted at the end of the shorter branches, thus making up the difference between the number of significant coordinates in a shorter branch and the number in the longest branch in the branching structure. The length of a branch is measured by the number of significant coordinates that it contains.

As just mentioned, loops are considered to be a special type of branching structure in which the end of each branch that does not break out of the loop must return to the root of the branches and (possibly nondeterministically) choose another branch into which to pass. Virtual significant coordinates are placed at the end of each branch (even those that do break out and exit from the loop) as needed following the same rule applied to any branching structure.

The state space is represented, as in SPIN, as a product of geometric components (distinguished from one another by the process control point, or program counter, information) and the discrete set of values of all data variables, and a geometric state space ignoring the values of data variables is constructed. The presence of conditional branching and loop iteration represents the greatest challenge in creating a partition.

5. CORRECT DEADLOCK DETECTION

5.1. OVERVIEW

This section discusses some correctness properties that apply to the inductive, cross-sectional, and progress shell partitioning methods. The essence of the initial problem solved by Goubault has already been stated implicitly. How can a state space be partitioned into components to produce a coarser, reduced digraph such that one can detect deadlock by exploring the digraph instead of the original, larger state space? That is, how can one be sure that deadlock will be detected in a reduced space *if and only if* it is actually possible in the larger space? This issue is rigorously addressed here.

The strategy for more efficient yet correct deadlock detection that is the subject of this dissertation relies on two major reductions. The first of these reductions concerns a partitioning method for decomposing a state space into components, and the second concerns a method of using that partition to guide a search through the state space. Therefore, those reductions and their correctness properties will be discussed in this section. A condensed treatment of this topic is presented in a paper that has recently been submitted to a conference [24].

Certainly an exhaustive search of the state space through all enabled transitions will find any reachable state where deadlock might occur, but such a search can be very inefficient. This problem is the motivation for component-based search. However, component-based search is insufficient when computational statements are part of a model process, for transitions between states inside the components must be executed when computations with data variables are involved. The first attempt at state search based on component-based search is called pessimistic state search, because it may include more transitions than are necessary in order to be sure that deadlock is correctly detected or that deadlock-freedom is correctly verified. An optimization is possible to reduce the number of transitions used, and this optimized version of component-based search is called optimistic state search.

The next subsection explains why a component-based search is correct. That is, if transitions were taken directly from component to adjacent component, then eventually a maximal node (in the digraph of components) that indicates deadlock would be found if and only if deadlock can occur in the underlying state transition system. It is because of that result that a state space search can be based on the partition, and this strategy has been discussed in previous work by Goubault et al. Consider any reachable state, x, in the state transition system. Essentially, what is needed for a component-based search to be valid is that the component of x is reachable by an admissible path in the associated digraph.

5.2. COMPONENT-BASED SEARCH

Let X be the state transition system of a concurrent program, and consider the associated progress graph $G(X)$ as having the structure of a locally partially-ordered space. Let $P(G(X))$ be a partition of the progress graph such that the components are rectangular and have boundaries whose faces are perpendicular to the axes. To address the question of when the partition can be used for accurate deadlock detection, the transitions between the components are defined according to when one component is immediately in the future of another. In other words, if the two components share a face of dimension one less than the number of processes, they are said to be neighbors, with the direction of the transition taken to be the direction of program progress. This definition is valid even when loops are permitted, because of the local partial-order.

A maximal node in the digraph represents a component in the partition from which there are no outgoing transitions; therefore, its immediate future is entirely forbidden. For a partition to be useful for deadlock detection, it must have a certain property relating maximal states in the state transition system to maximal nodes in a corresponding digraph. In the context of a partitioning algorithm, a partition is said to possess a property if every partition that it outputs also possesses it.

Property 47. A partition of a progress graph possesses the *correct deadlock detection property for components* if the program that the progress graph represents can deadlock if and only if the search of the component graph associated with the partitioned progress graph indicates that deadlock is possible.

Property 48. A partition of a progress graph possesses the *trace representation property for components* if every maximal trace fragment is represented by a maximal path specific to that trace.

Property 49. A partition of a progress graph possesses the *bijective type correspondence property for components* if there exists a bijective assignment of a type class of maximal paths (with admissible type) in the component graph to each dihomotopy class of maximal dipaths.

Goubault and Haucourt [15] claim the following result, the proof of which is omitted here, in Section 3.3 of their paper.

Theorem 50. *For any progress graph associated with a concurrent program, there is a bijection between maximal trace fragments of the program and dihomotopy classes of maximal dipaths in the progress graph.*

Theorem 51. *If a partition of a progress graph possesses the bijective type correspondence property for components, then it also possesses the trace representation property for components.*

Proof of Theorem 51. Theorem 50 asserts that every maximal fragment corresponds to a particular dihomotopy class of maximal dipaths. Then, by Property 49, it is assigned a type class of maximal paths. □

Theorem 52. *If a partition of a progress graph possesses the trace representation property for components, then it also possesses the correct deadlock detection property for components.*

Proof of Theorem 52. Every maximal trace fragment is represented by a maximal path in the component graph. This applies in particular to maximal trace fragments that begin at the initial point, and of those, successful ones correspond exactly to successful admissible maximal path type classes. □

The following lemma says that for any maximal admissible path in a digraph corresponding to an (appropriately) partitioned progress graph, there is a maximal dipath that maps to it.

Lemma 53. *Let D be the set of maximal dipaths in a progress graph, let A be the set of maximal admissible paths in the associated digraph (from the box- or L-shaped partition, or Progress Shell Decomposition), and let P be the set of all maximal paths in that digraph. It will be shown that f is well-defined and surjective (whereas g is clearly injective), and the upper triangle in Figure 5.1 commutes, with $h = g \circ f$.*

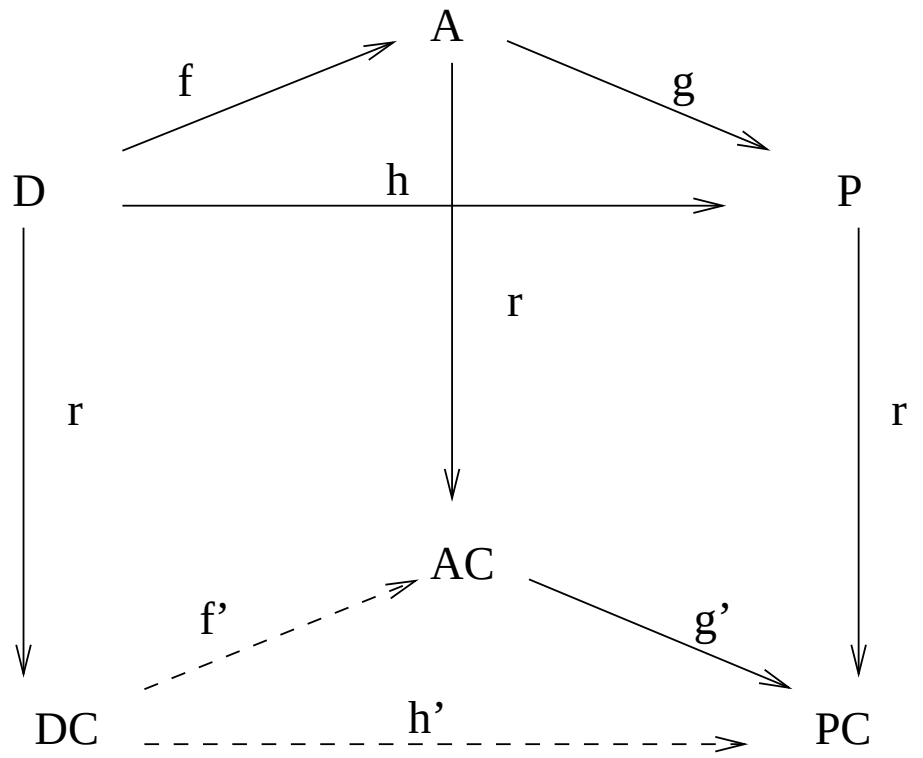

Figure 5.1. A commutative diagram relating the operations of passing from dipaths to paths and from dipaths/paths to classes induces the bijection f'.

Proof of Lemma 53. First of all, suppose that p is a maximal dipath in a progress graph with forbidden rectangles. If p is valid and successful, then it clearly maps to a maximal path in the associated digraph. There are two other (non-exclusive) possibilities. If p is maximal but not successful, then it must reach a point of deadlock x. Because p must approach x to an arbitrary proximity, there is only one region R that contains points on p for parameter values that are arbitrarily close to 1. Therefore, R must be shown to be maximal. In order for deadlock to occur at x, it must be on the boundary of two forbidden rectangles, one above it and one to its right. In higher dimensions,

other interesting arrangements of forbidden columns must be considered that can create deadlock, but a similar argument applies. Let v and w be the minimal (lower left) corners of the significant rectangles corresponding to these forbidden rectangles, as illustrated in Figure 5.2.

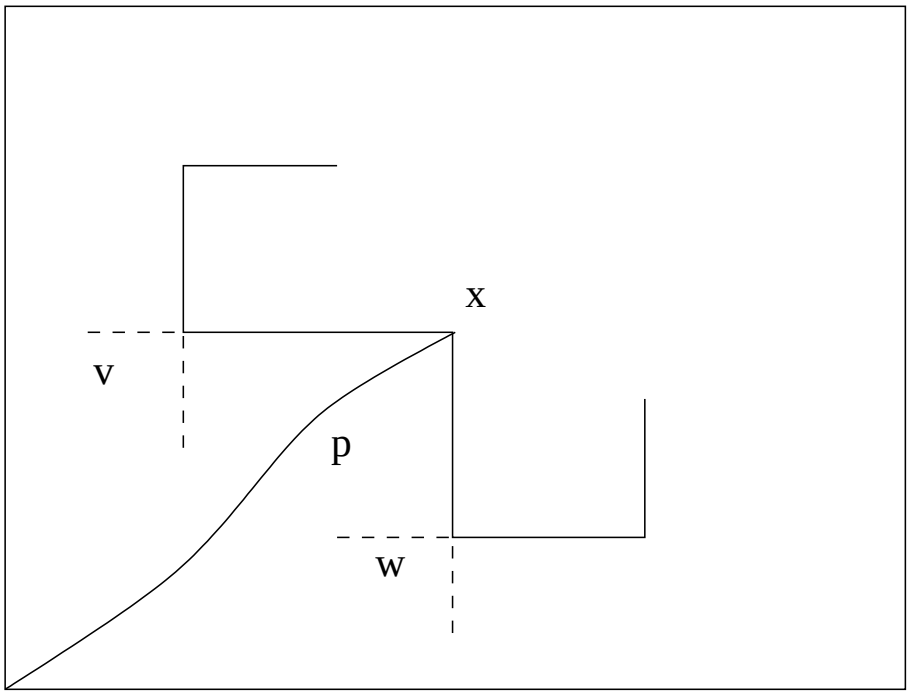

Figure 5.2. Here, p must go between v and w to a maximal region. Since every minimal corner of a forbidden rectangle is a point of intersection of two perpendicular line segments (of the partition) passing through it, p ends in a maximal rectangle no matter what other partitioning is done by the algorithm.

Consider the rectangle C formed with v as the upper left corner and w as the lower right corner (and with x as its upper right corner). R must be contained inside C, so it is maximal because all regions of the partition (including R) are rectangular. The possibility of p being invalid can be handled by similar arguments.

In case of more than two processes, let x be a point of deadlock. Consider the pairs of processes for which there is a forbidden rectangle blocking the progress (forward extension) of the path p at x. For each axis in which the coordinate of x is not the same as

the terminal point of the corresponding process, there is at least one pair with a forbidden rectangle blocking progress in that direction.

Define F_i to be the set of axes of the other processes in such pairs particular to the axis i. Renumber the processes so that F_1 is non-empty. Thus, there is at least one axis in F_1. Also, because x is a point of deadlock and forbidden rectangles are in the interior of a two-dimensional progress graph, there is an interval (with 0 as the lower bound and the coordinate of x as the upper bound) corresponding to that axis such that another forbidden rectangle in some (perhaps different) two-dimensional progress graph blocks progress at x. Renumber the processes other than the first one chosen so that F_2 is the set of axes corresponding to the axis in F_1 where the first interval was found. One may continue to choose axes and intervals until a cycle occurs, because the number of axes is finite.

Replace the zeros by new lower bounds using the forbidden rectangles that appeared (as axes and intervals were chosen) in the two-dimensional progress graphs for the axes in the cycle. For each such forbidden rectangle, use the second coordinate of its minimal corner (see Figure 5.3). Observe that the point of deadlock is the maximal point of a cuboid formed by the product of the new intervals, and this cuboid contains a maximal component of the partition (as in the two-dimensional case). The path p will necessarily enter this maximal component, because it reaches the point x; therefore, there is a corresponding maximal path in the digraph as well. □

This next lemma says that any two representatives of a dihomotopy class of maximal dipaths map to maximal admissible paths that have the same type. It also says that every admissible type class of paths has a unique representative dipath class that maps to it. A corollary is that finding maximal admissible paths in a reduced digraph representing the progress graph is sufficient to do trace-consistent model checking, especially deadlock detection.

Lemma 54. *Passing to (dihomotopy) type classes, let DC, AC, and PC be equivalence classes corresponding to the sets of dipaths, admissible paths, and all paths. The entire diagram in Figure 5.1 commutes, with $h' = g' \circ f'$; moreover, f' is bijective.*

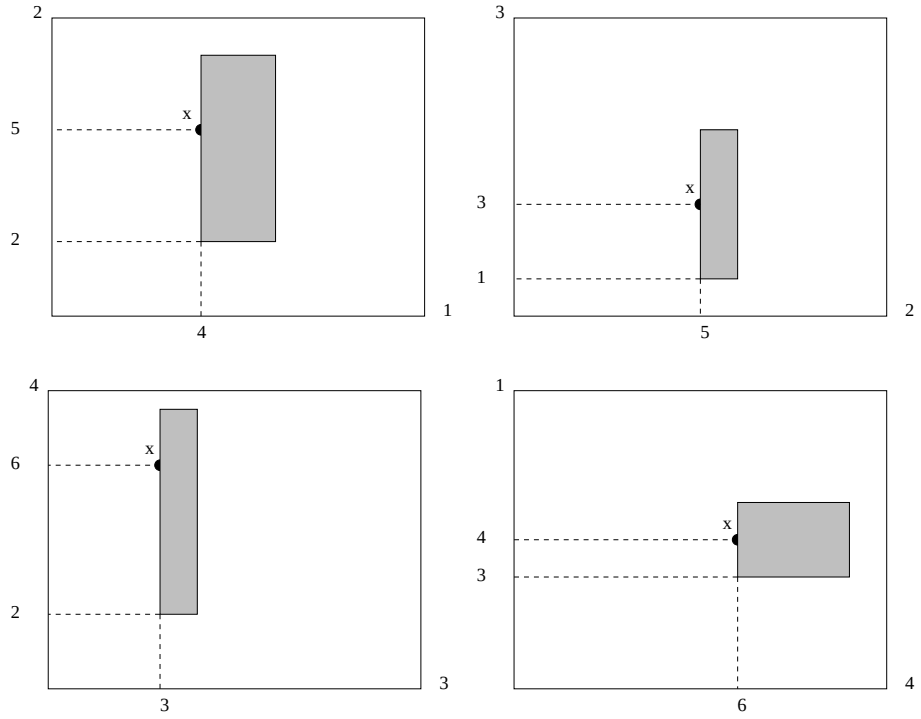

Figure 5.3. Here, $x = (4, 5, 3, 6)$ is a point of deadlock and the maximal corner of the cuboid $[3, 4]$x$[2, 5]$x$[1, 3]$x$[2, 6]$.

Proof of Lemma 54. It must be shown that f' and g' are well-defined, so $h' = g' \circ f'$ is also well-defined. The map g' is an inclusion, clearly well-defined. To show that f' is well-defined, suppose that p_0 and p_1 are two dipaths in D that represent the same class p in DC. Then $f'(p)$ can be defined as $r_A \circ f(p_0)$ or $r_A \circ f(p_1)$, which are equal by Definition 31. Therefore, since r_D is surjective, f' is well-defined.

It may be observed also that

$$h' \circ r_D = g' \circ f' \circ r_D = g' \circ h \circ f = r_P \circ g \circ f = r_P \circ h,$$

so the rectangle in the front of the diagram commutes.

Finally, one needs to show that f' is surjective and injective, so f' is a bijection. The surjectivity of f' follows from the surjectivity of f and of r_A. As for injectivity, let $s, t \in A$ represent an admissible maximal path type class $[s] = [t] \in AC$, and suppose that $p, q \in D$ represent dihomotopy type classes $[p], [q] \in DC$. Also suppose that $f(p) = s$ and that $f(q) = t$, so $f'([p]) = f'([q])$ by the commutativity of the diagram. Corollary 46

implies that p and q are dihomotopically equivalent, so $[p] = [q]$. Therefore, f' is in fact injective. □

Corollary 55. *The cross-sectional partitioning methods described in the previous section (box- or L-shaped, or Progress Shell) possess the bijective type correspondence property for components; therefore, by Theorem 51, they also possess the trace representation property for components.*

Theorem 56. *The cross-sectional partitioning methods described (box- or L-shaped, or Progress Shell) possess the correct deadlock detection property for components.*

Proof of Theorem 56. Combine Corollary 55 and Theorem 52. □

The following result, the proof of which is omitted here, is another result that is presented in Section 3.3 of the paper of Goubault and Haucourt [15].

Lemma 57. *Goubault's inductive partitioning method possesses the bijective type correspondence property for components; therefore, by Theorem 51, it also possesses the trace representation property for components.*

Theorem 58. *The inductive partitioning method for progress graphs possesses the correct deadlock detection property for components.*

Proof of Theorem 58. Combine Lemma 57 and Theorem 52. □

5.3. STATE SEARCH

Given a partition as above, two search problems can be defined based on it. The first is called pessimistic, because, being overly cautious, it most likely has a space with more transitions than are necessary to detect deadlock. The second search problem is called optimistic because it is assumed to have a space with a more reasonable set of transitions. A pessimistic state search of a partitioned progress graph takes at least one transition from any node to each of its neighbors in the associated digraph. An optimistic state search rejects transitions to neighbors that are not minimal with respect to being in the future of the current component. That is, if a neighboring component can be reached by going through another neighbor, then the transition to the minimal

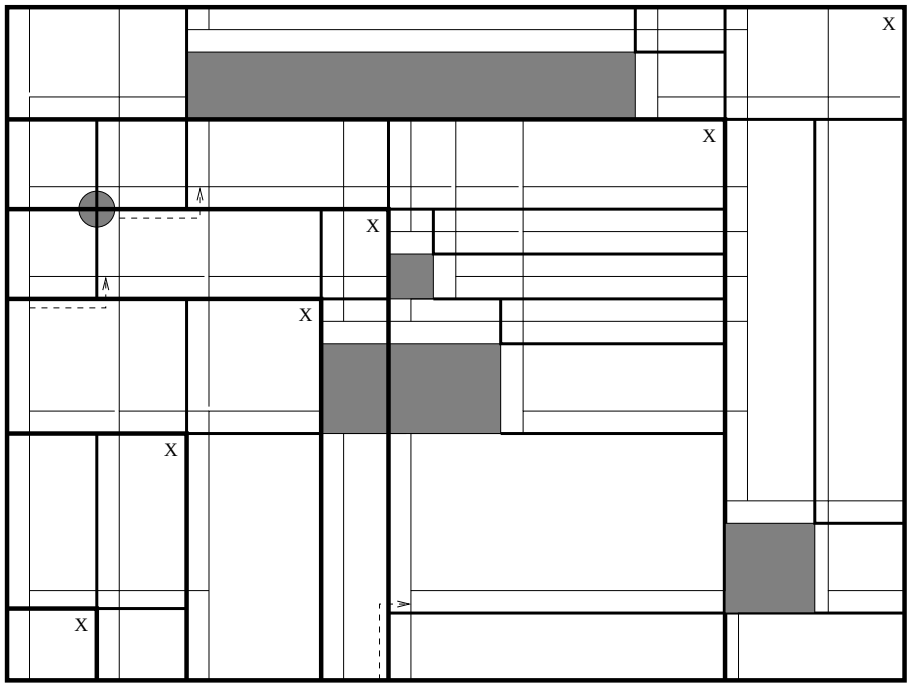

Figure 5.4. The Progress Shell Decomposition is given by the thick boundaries. Middle corners are marked by an X at the bend in the shell boundaries. Optimistic Search includes the solid path segments, whereas Pessimistic Search includes the dashed path segments as well.

intermediate neighbor is taken instead of the transition directly to the originally-referenced neighbor – see Figure 5.4.

The partition guides the choice of transitions to be used during the search of the state space. Thus, a partition must be sufficiently fine to guide the search correctly so that deadlock is detected if and only if it is actually possible. It would also be interesting to know when a partition is optimal with respect to minimizing the duration of the search (in terms of either the number of states visited or the number of transitions executed). A good approach would begin by discovering when a partition is *minimal* with respect to refinement: when is it minimally fine but leads to correct deadlock detection?

The analogues of the previously defined properties are now easy to formulate for state sequences. In this context, the properties are defined for the pessimistic case, but the definitions remain valid for the optimistic case if the word optimistic is substituted in place of the word pessimistic.

5.3.1. Pessimistic State Search

Property 59. A partition of a progress graph possesses the *pessimistic correct deadlock detection property* if the program that the progress graph represents can deadlock if and only if pessimistic search of the state graph associated with the partitioned progress graph indicates that deadlock is possible.

Property 60. A partition of a progress graph possesses the *pessimistic trace representation property* if every trace fragment of the program that the progress graph represents is represented by a path in the pessimistic state graph associated with the partitioned progress graph, and if every maximal trace fragment is represented by a maximal path that is specific to that trace.

Property 61. A partition of a progress graph possesses the *pessimistic bijective type correspondence property* if there exists a bijective assignment of a type class of maximal paths (with admissible type) in the pessimistic state graph to each dihomotopy class of maximal dipaths.

Theorem 62. *If a partition of a progress graph possesses the pessimistic bijective type correspondence property, then it also possesses the pessimistic trace representation property.*

Proof of Theorem 62. This theorem follows directly from Theorem 50. □

Theorem 63. *If a partition of a progress graph possesses the pessimistic trace representation property, then it also possesses the pessimistic correct deadlock detection property.*

Proof of Theorem 63. This is a simple generalization of Theorem 52 applied to state sequences, and it follows because attention can be restricted to trace fragments that begin at the initial point. □

Theorem 64. *The progress shell partitioning method possesses the pessimistic correct deadlock detection property.*

Proof of Theorem 64. The only observations necessary for a proof of this theorem (using the proof for Theorem 56) are that in this case every reachable maximal component

is visited and that once such a component is visited, its maximal state will be reached also. In other words, this theorem is nearly a corollary of Theorem 56. □

5.3.2. Optimistic State Search As complete correctness is the goal of the software discussed here, it is necessary to justify the decisions that are made during the optimistic state space search that lead to an improvement in efficiency. For example, this state space search is considerably sparser in terms of paths explored than an exhaustive search would be, because many transitions are refused. An overview of the method for deciding which transitions to take and which to neglect follows, and the important assumptions for correct deadlock detection of optimistic state search are enumerated as follows.

1. Each time that a component is entered, transitions may branch off in each direction parallel to an axis and follow a path in those directions to the forward boundary of the current component.

2. When the search encounters a forward boundary of a component, and the next component adjacent in the direction that was being taken is not forbidden, it is not necessary for the search to branch: it may continue forward one more step into the new component.

3. When an adjacent component is forbidden, the search path may simply take the minimal length path around the forbidden rectangle in the two-dimensional progress graph where the blockage occurs (if such a path is possible).

4. The search may repeat the tactic of the previous item in case multiple consecutive blockages are encountered in the two-dimensional progress graphs.

The search begins at the origin – the initial point of the entire state space. From this state all possible directions representing valid transitions are included in the search paths to be explored. In fact, any time during the search that a new component of the partitioned state space is entered, a certain rule will be followed. That rule is that each possible direction orthogonal to the axes will be explored as far as the far forward boundary of the new current component. In the code this is accomplished by a loop through the different directions that calls a function called big_move.

This big_move function begins by branching into all possible directions, but then restricts the exploration to straight paths through the current component, so it will generally refuse to take some possible transitions. Why is this stategy valid, one might ask? The answer is that straight paths make the minimum progress (zero) in every direction except in the direction chosen: zig-zag paths through the interior of components are avoided because the partition itself is fine enough that they become unnecessary. This is the first claim that needs further justification.

Next, whenever a search path encounters a forward boundary of a component, the search calls a function called tiny_move, which simply takes one more transition continuing in the same direction, in order to cross into the next component (if it is not forbidden). Only when the adjacent component is forbidden is a change of direction permitted to occur by the search strategy. Thus, tiny_move will generally reject some possible transitions. The validity of that decision is the second claim that needs justification, and it is closely related to the first claim.

When a forbidden component is encountered at a forward boundary of the current component, the search path changes direction to go around the forbidden component; however, it need not branch in every direction, nor is it necessary or even appropriate to continue as far as the next component after it does branch. The search only needs to determine the two-dimensional progress graph that contains the forbidden rectangle that is blocking its path forward, and travel inside that plane forward just past the corner of the forbidden component and then turn to take a transition in the direction that it was originally traveling when it was blocked. This is the third claim that needs justification.

In case there are repeated blockages, the search may continue the process described in the previous paragraph until egress is found or until a point of deadlock is reached. This is the fourth claim requiring justification.

The reason why the optimistic state search strategy is a correct one is that the least progress is made toward adjacent components; therefore, every possible transition between components is taken. Minimal transitions between components suffice for deadlock detection because of the way the progress shell decomposition creates maximal components containing the states where deadlock may occur.

Property 47 is the basis for the justification of the first claim, for there is an admissible path to a maximal node in the digraph of a state transition system if and only if a point of deadlock exists. To complete the justification, observe that the cuboidal shape of the components allows one to choose a rectangular dipath with changes in direction only at the minimal point of each component or on its forward boundary to represent the admissible path. Moreover, the only time that a branch is necessary at the forward boundary is when a continuation into the next component is blocked by a forbidden column,

Thus, if possible, the optimistic state search takes *one* additional transition into the next component and repeats the procedure. If the component happens to be forbidden, then it takes the shortest path around it to a component in the direction in which the path was originally headed. Observe also that when it is necessary for a dipath to change direction due to a blockage, that direction is blocked by a single forbidden column represented by a single forbidden rectangle in a single two-dimensional progress graph, as shown in Figure 5.5. There is no need for additional branching during this maneuver, because only binary semaphores are permitted.

Lemma 65. *The situation depicted in Figure 5.5 is impossible.*

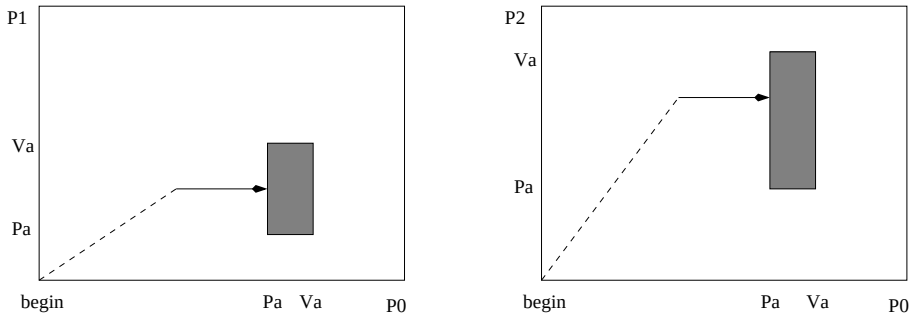

Figure 5.5. In this three-dimensional progress graph, the search reaches a state that is blocked in the same direction by two different forbidden rectangles that occur in different two-dimensional progress subgraphs.

Proof of Lemma 65. Process P_1 and P_2 cannot be in the same critical section for the same semaphore simultaneously, because only binary semaphores are allowed. □

When a forbidden rectangle blocks one direction, progress in only one direction orthogonal to it will relieve the blockage. If progress in that second direction is subsequently blocked before the first blockage is relieved, then again there will be only one third direction that can relieve the second blockage. If there is ever a need to go forward in a direction that is still blocked, then deadlock is indicated, but otherwise eventually an egress will be discovered for each blockage in a last-in-first-out (LIFO) manner. A shortest path strategy for skirting the blockages is appropriate because no admissible paths through the component digraph will be missed if that strategy is used. The following theorem summarizes the discussion of optimistic state search.

Theorem 66. *The reduction from pessimistic search to optimistic search and the correctness of the DRUiDD search procedure is justified.*

6. APPLICATION: DRUIDD

Version 4.3.0 of SPIN, from www.spinroot.com and locally-developed code is used for this work. In the ensuing discussion, each algorithm developed for DRUiDD references its underlying theory.

6.1. SPIN CODE STRUCTURE

6.1.1. SPIN Parser

The fundamental unit of a PROMELA model is a statement within a process. Statements give rise to control points in processes and to state transitions between control points. Therefore, discussion of the SPIN parser begins with a brief explanation of how statements are converted into transitions. SPIN recognizes statements by their actions: message-passing, assignment, conditional expression, etc. DRUiDD (Dihomotopic Reduction Used in Deadlock Detection) was tested on models containing some of these types of statements, and the framework that already exists for the parser was used. The same data structures and functions that convert PROMELA statements into a transition table in the file pan.t were modified as well.

SPIN recognizes statements and executes certain actions upon encountering one within a model being parsed. The function that contains the code to be executed is called "put_seq". This function is called by the "put_proc" function, and it calls several other functions either directly or indirectly. The most important of those for the current work were "put_el" and "put_sub", which together with put_seq do most of the work of creating the transition table that is used during the search of the state space of the PROMELA model. The primary modifications that were made involved a slight alteration of the transition data structure in order to encode the accesses to global variables and semaphores by statements.

6.1.2. PAN Verifier

SPIN produces five files from a single model, and they are pan.h, pan.c, pan.t, pan.m, and pan.b. In order to do any model checking with SPIN, one first parses a model, then compiles an executable (called a verifier, and usually named "pan") from the pan.* files; then one runs the executable. The pan.* files contain all of the information about the model that is necessary to perform the desired model checking

verification. In particular, they must encode the model's statements in pan.t and pan.m. The difference between these two files is that pan.t primarily encodes the flow graph information for the processes of a model, and pan.m encodes the effect of the transitions in the flow graph (corresponding to statements in the model) based on the effect that the statement has on the data structures declared within the model. Thus, pan.t contains more information about the topology of the program, whereas pan.m contains more information about the data variable transformations.

The other files (pan.h, pan.c, and pan.b) also require some explanation. The file pan.b encodes the backward moves needed when an execution trail is being taken off the stack to permit exploration of another execution path (the verifier uses depth-first search by default). The other two files, pan.h and pan.c, contain what one expects in a C program – variable declarations and function implementations. Most importantly, pan.c contains the functions that implement the depth-first search of the state space.

Holzmann [25] discusses the basic features of the implementation of PAN; therefore, the following offers only a very condensed description. The function of primary interest is called "do_the_search". Within it, function calls were added to do the geometric preprocessing of the state space for DRUiDD search. The search itself is handled by a function call to the "new_state" function. According to Holzmann, it uses pseudo-recursion; it employs goto statements to carry this out by repeatedly returning to a section of code that is labeled *Down*, where an execution path is extended by one transition. From time to time, however, because by default the function implements a depth-first search, it visits a section of code that is labeled by the term *Up*, where the execution path is undone by backtracking.

The modification presented here of the search procedure sought merely to insert some extra statements into the new_state function in order to carry out the optimistic guided search based on the progress shell decomposition of a state-space. These modifications became too complex; therefore, the new_state function was rewritten using the original as a basis. Ordinary recursion was used due to time constraints, and the function proved much easier to implement than the original version would have been.

6.2. DRUIDD STRUCTURE

The three correctness claims (and corresponding theorem references) for DRUiDD search are enumerated here for added clarity:

1. Component-based search based on the progress shell decomposition is as correct as exhaustive search when there are no data variables (Theorem 56).

2. Pessimistic state search is an appropriate generalization of component-based search when there are data variables. Thus, if component-based search is correct when there are no data variables, then so is pessimistic state search when data variables are considered (Theorem 64).

3. Optimistic state search is equivalent to pessimistic state search for deadlock detection (Theorem 66).

The functions for geometric preprocessing of the state space by DRUiDD were put into separate files to be made available by #include directives in the pan.* files.

6.2.1. Analysis The first of the supplementary DRUiDD files is called "dr-analyze." Its job is to determine the topological and geometric structure of the state space so that it can be correctly partitioned into components later.

Conditional branching and looping structures are treated as being the same at this stage, and what is needed is information about the nesting of these structures within each process and about the length of the segments that make up the branches within each structure. Each branching structure has a beginning and end, and these must be identified correctly. This analysis determines the topology of the flow graph of each process.

Semaphore locks and unlocks determine the minimal and maximal corners of forbidden rectangles in the two-dimensional progress graphs that are of fundamental importance for the partition and search. One must correctly form pairs consisting of one lock and one unlock statement to know where the critical sections for each semaphore are within each process. Not only that, but it is also necessary to match critical sections (specific to each particular semaphore) between pairs of processes to locate the forbidden rectangle corners in each two-dimensional progress graph.

As with semaphore locks and unlocks, read and write accesses of global variables can create significant points within each two-dimensional progress graph. This is because it does matter sometimes whether an execution path goes above or to the right of a point where two writes or a read and a write might occur simultaneously in the two processes being combined (Lemma 37). Therefore, a list of variable accesses for each assignment and conditional guard was maintained in the transition table, and it is at this stage of the preprocessing that this information is especially useful.

6.2.2. Partition After the analysis of processes and pairs of processes is complete, the decomposition of the two-dimensional progress graphs can begin. The progress shell decomposition is obtained by partitioning the progress graph using two phases – initial partition, in which the progress shells are computed, and secondary partition, in which the individual shells are divided into rectangular components.

The file "dr-partitioninit" contains the definitions of the functions used in the initial partitioning. The major task to be completed at this stage is the determination of the middle corners of the boundaries between consecutive progress shells; however, it is also important that the shells be stored in a data structure that allows easy traversal through the shells from the initial point of each two-dimensional progress graph to the terminal point. More precisely, a data structure is defined that links consecutive boundaries to form a progress shell, and then information about the adjacency of consecutive progress shells is also maintained (Theorem 56). This approach facilitates the completion of the decomposition during the secondary partitioning stage.

In order to locate the middle corners, one finds matching significant coordinates in the flow graphs of the two processes that are being considered for the current two-dimensional progress graph. One way to understand this procedure is to imagine that each significant coordinate is assigned a numerical value depending on its distance from the initial point of the process in which it is located, where distance is marked off incrementally by the significant coordinates themselves – not by every control point, which may or may not represent a significant coordinate. Then coordinates in the two processes with equal numerical values are matched and paired together as middle corners.

In the case of branching, ambiguity can arise *after* the branching structure is passed if two or more branch segments have differing numbers of significant coordinates within them, so it is agreed that the enumeration begins after the branching by using the maximum number of significant points contained in any of the branch segments in that particular structure. To make up the difference between the different branch segments, and to ensure continuity in the enumeration along the different paths through the structure, virtual significant points are created as needed to equalize the number of significant coordinates. This method is justified by consideration of the need to match up significant coordinates in both processes in the progress graph: gaps in the enumeration would create difficulties, but by using virtual points to equalize the branch segments, this issue is resolved successfully.

The functions used in the secondary partitioning are defined in the file called "dr-partitionsec". Here one uses the location of significant points and rectangles to complete the decomposition of the progress graph into rectangles that can easily be transformed into a digraph for use during the search of the state space of the model being verified by DRUiDD. The first step is to partition each progress shell into an upper-left, a middle, and a lower-right rectangle. Next, a function called break_rect_byrect is used to subdivide each of those rectangles depending on the positions of nearby significant points and rectangles within the progress graph. Pseudocode is given below for this algorithm, as well as an illustration of how the geometry affects the decomposition of the rectangle that is passed in as an argument (Figure 6.1).

```
/* General Helper (Rectangle) */
/* break_rect_byrect: Takes a target and a breaker rectangle. */
/* The intersection of rectangles produces a decomposition.*/
rectangle * break_rect_byrect(rectangle * target,
                       rectangle * breaker, int forbid)
{
  if(!target)  return NULL;
  if(!breaker) return target;
  target->nxt = break_rect_byrect(target->nxt,breaker,forbid);

  if(breaker contains target)
  { if(forbid == 1)  target->isForbidden++;
    /* No breaks will occur within this region*/
    if(forbid != 2)  return target;
```

```
    else              return target->nxt;
  }
  else if(any part of the breaker is inside the target)
  { for(each rectangle numbered zero through six)
    {  Determine how the corners of the target are situated
         with respect to the corners of the breaker.
       Assign the rectangle with number in 0..6 the
         appropriate coordinates as its corners. In general,
         these seven rectangles will surround the breaker
         and be inside the target.
    }
  }
  Relate all the broken bits into a new linked list.
  Join the new linked list to the former one (from
    which the target rectangle came).
}
```

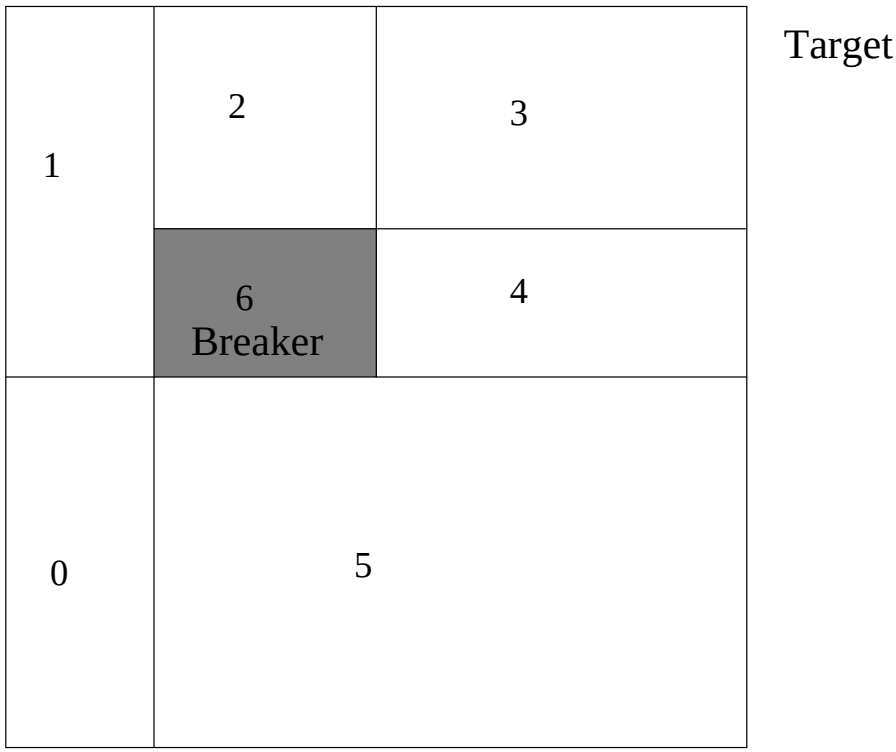

Figure 6.1. If the breaker is entirely inside the target, then the seven rectangles shown will be formed. Otherwise, there will be fewer, depending on how the corners of the target and breaker rectangles are spatially related in two dimensions.

One obtains a list of rectangles comprising each progress shell. It is important to remark here that the secondary partitioning is appreciably more complicated than the

initial partitioning because of the possibility of branching structures in either process of the two-dimensional progress graph. The added complexity arises particularly when two progress shells overlap in the region of the progress graph that represents a section of a process prior to the branching structure.

6.2.3. Digraph With the progress graphs corresponding to each pair already decomposed into rectangles, the next step is to store each one in a digraph (in the traditional sense) using adjacency lists. Functions for this task are contained in the file "dr-graph". Forming an adjacency list representation of the digraph is accomplished by first determining the directed edges among the rectangles of each progress shell and then among the adjacent progress shells, which is straightforward.

6.2.4. Exploration The file "dr-explore" contains the definitions of the functions for state space exploration. This exploration is performed using optimistic state search based on its partition from the progress shell decomposition. A summary of the algorithm is given in the paragraph below.

A function called "move" is called repeatedly during the exploration. It takes the execution path from one N-dimensional component to the next by calling on two helper functions. The first helper function is called "big_move" and the second "tiny_move." Big_move is used to advance the current state through a component that has just been entered to its forward boundary if it is not already there. Tiny_move is used then to advance one transition forward into the next component, after which the move function is called again. The move function then calls big_move within a loop over the various processes in order to allow all direction to be explored. Branching within a single process is accomplished here by an advance to the control point that is the root of the branching structure and then to the guards that represent different optional branch segments within the structure.

6.3. CHALLENGES

6.3.1. Design The DRUiDD software was primarily designed and implemented by a two-person team consisting of the author and an undergraduate assistant, and many aspects of the design were worked out as the software was implemented because of the need for communication about how best to cope with various issues.

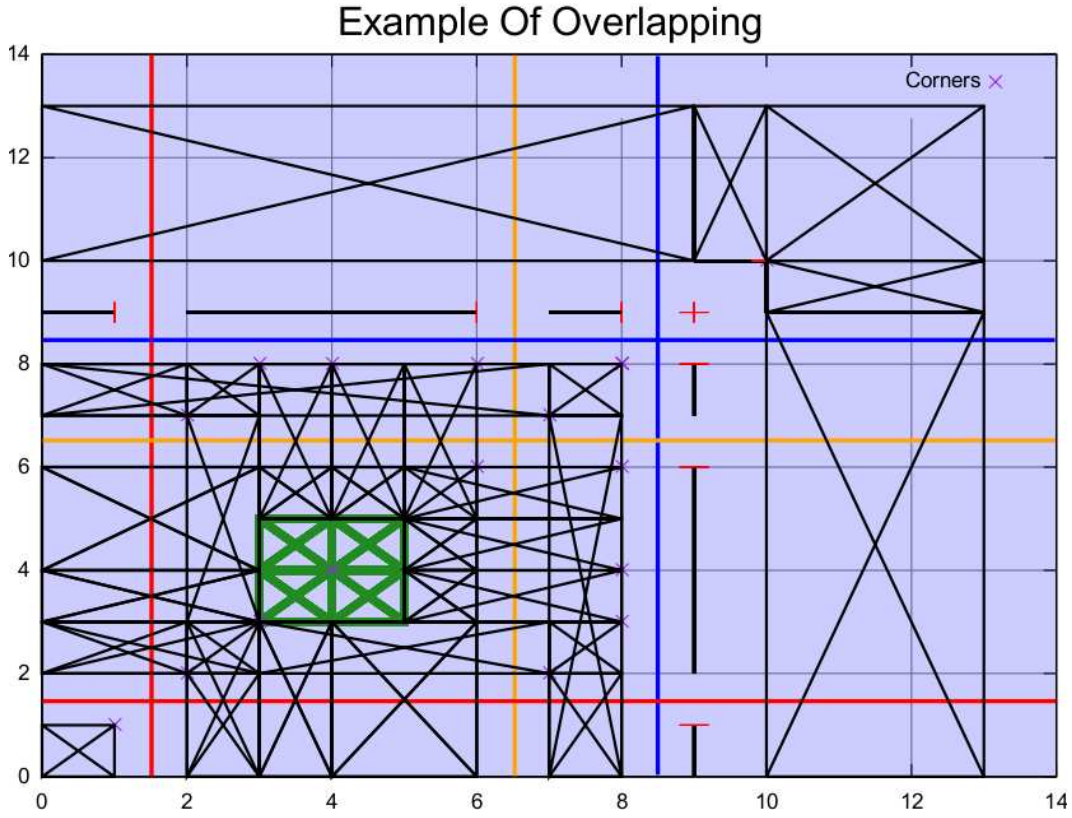

Figure 6.2. Overlapping of components was detected using a visualization tool. Diagonal lines helped to identify the interiors of the components.

6.3.1.1. Partition As has already been described, the initial partition requires an equalization procedure for each branching structure in a process. Working out the exact method for accomplishing this was an interesting design issue. In designing the secondary partition algorithms, a significant problem arose because of the potential overlap of progress shells when a process contains branching structures. To make this more clear, one can represent a process that does not have branching structures along the vertical axis of a plane and represent a process with one branching structure that has two branches along the horizontal axis. Thus, when progress shells are drawn back toward the left boundary of the progress graph from middle corners that occur in each of the branches at the same height, there will be some overlap near the left boundary, as illustrated in Figure 6.2.

This problem was overcome by truncating the progress shell that arose from the *second branch* at the beginning of that branch.

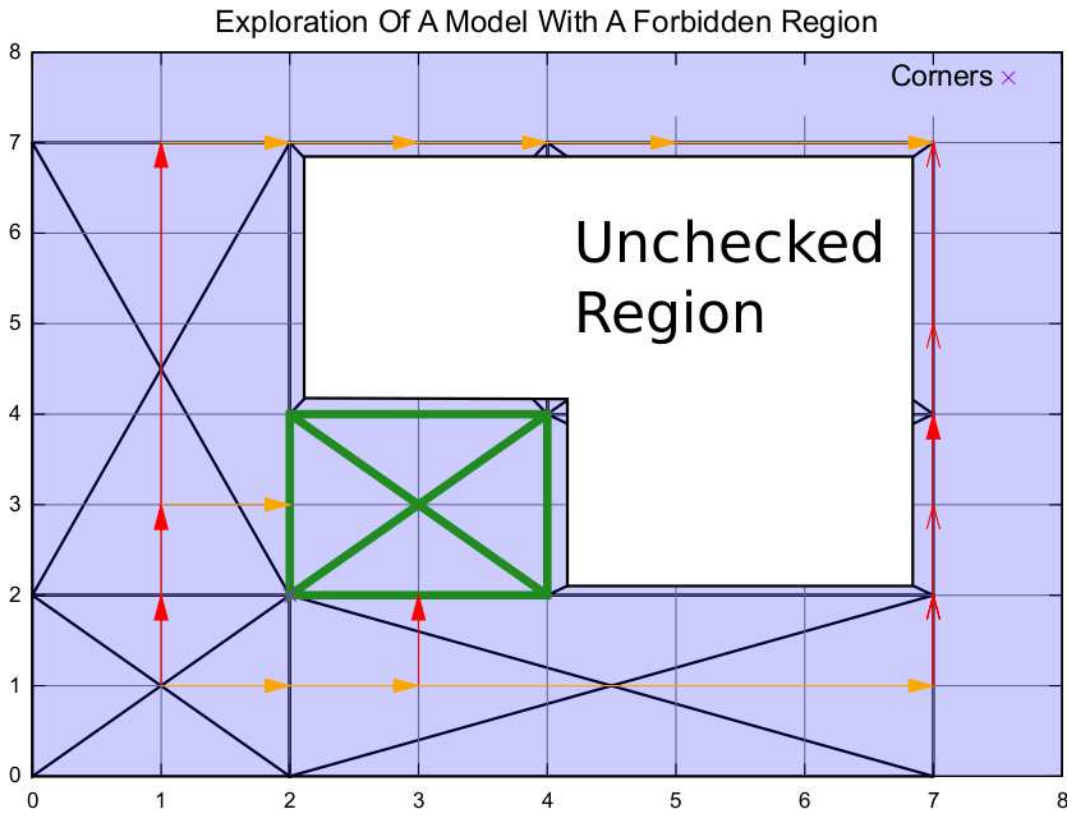

Figure 6.3. If the search path does not wrap around the forbidden rectangle, then some regions of the progress graph could be left unexplored, so a point of deadlock could be missed.

6.3.1.2. Search When a forbidden rectangle occurs on one forward boundary of a component that extends forward beyond the forbidden rectangle in the orthogonal direction, it is necessary that execution paths wrap around the forbidden rectangle as tightly as possible when they are blocked in the first direction. That is, suppose a path is going to the right and the tiny_move function cannot advance the path any further to the right, but a move upward is possible. One wants the shortest move upward to take place before the path tries to go to the right again, because otherwise some legitimate paths might not be explored – a conservative approach to exploring the space. This is correct essentially for the same reason that optimistic search (rather than pessimistic search) suffices for deadlock detection (Theorem 66). The construction depicted in Figure 6.3 and Figure 6.4) illustrate the potential for overlooking a region where deadlock might occur if a less conservative strategy were adopted.

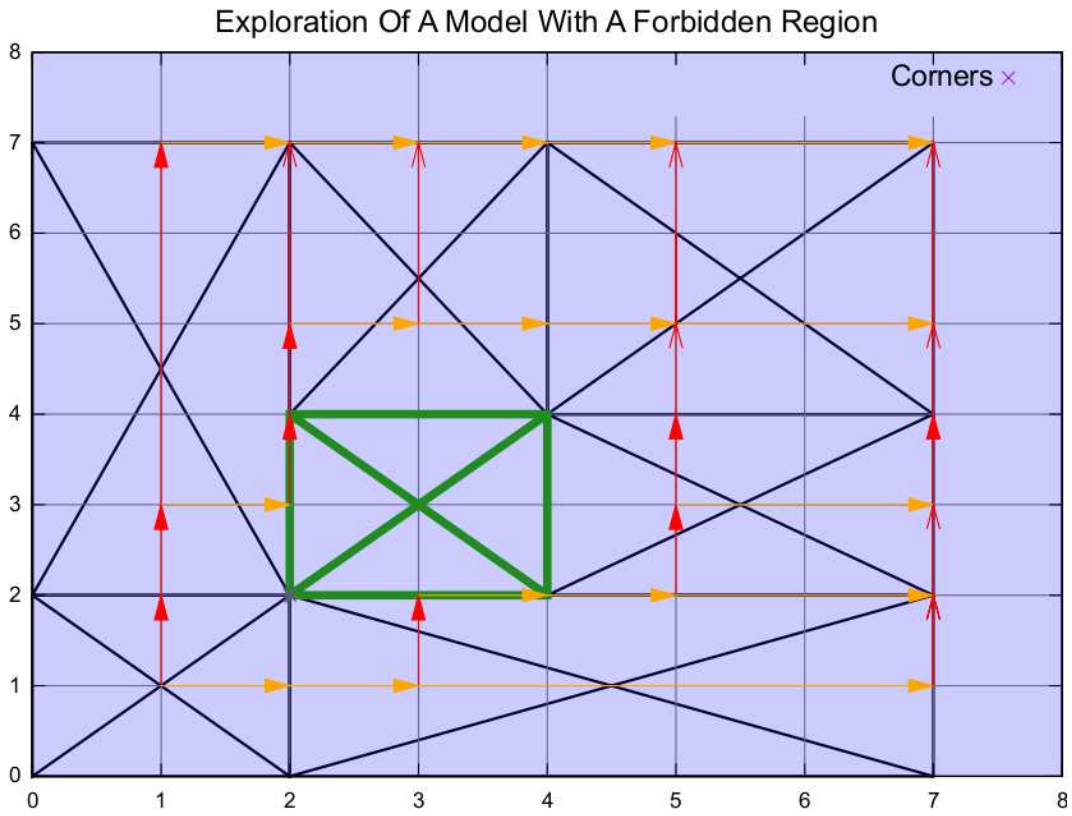

Figure 6.4. Correctly wrapping around forbidden regions ensures that reachable areas are explored.

To unify the way that the move function operated, zero-area (degenerate) components were introduced in the digraph. In other words, every state is considered to be inside an actual component in the digraph, even if that component has no absolutely area at all; for example, its left and right boundaries might coincide, so it is really just a vertical line segment.

6.3.2. Implementation

6.3.2.1. SPIN Parser and Code Generation The very first difficulty during the implementation was locating a reasonable place for preprocessing function calls within the pan.c file and then figuring out how to generate the code for those function calls inside one of the pangen files. Interestingly, the code generation is mostly accomplished by printing huge arrays containing lines of code, so it was easy to just insert the desired statements into the arrays as strings.

The first task that required some thought during implementation of the preprocessing procedures was how to capture and put into list form the semaphore operations and global

variable accesses within the individual statements of processes. Doing so was necessary for correct partitioning of progress graphs, and the problem was solved by modifying the transition struct to encode this information.

Also, the SPIN new_state function was rewritten (modeled on the original one) because of the need to utilize the geometric information about the state space.

6.3.2.2. Transition Sequence Out-of-sequence control points for branching structures caused some delays due to the author miscommunicating during explanation of the design. In trying to communicate the partitioning procedure, pictures were sketched that were true to the ideal geometry of progess graphs but not true to the way SPIN encodes transitions into a transition table. That is because the SPIN parser places the control points for *if* and *do* structures *after* the subordinate branches when enumerating the states and transitions in a process flow graph. Eventually, a better graphical representation was devised that allowed better communication between the author and the assistant who was doing most of the programming.

6.3.2.3. Correctness, Revisited The issue of the difference between the geometry and the transition table leads naturally to a discussion of the topic of the correctness of the actual implemented partition and search versus the correctness of the underlying theoretical geometric model of the state space search. The SPIN parser creates a transition table from the statements in each process, and branching structures are encoded in the table in a special manner. Usually, statements are encoded in a sequence corresponding to their occurence in the process model, but when there is a branch, this becomes impossible to do: there will necessarily be gaps that have to be jumped across during simulations or verification runs of the model. This is best illustrated with an example, as shown in Figure 6.5.

When the program counter arrives at the control point at the beginning of a branching structure, there can be several different branch segments that can be executed next, depending on which guards are open. The SPIN parser solves this issue by creating a linked list of transitions to the guard statements. Each guard is followed by the statements in its branch segment, and then there is a transition back to a point of union of the different branch segments, at which point the program continues on. Thus, there are

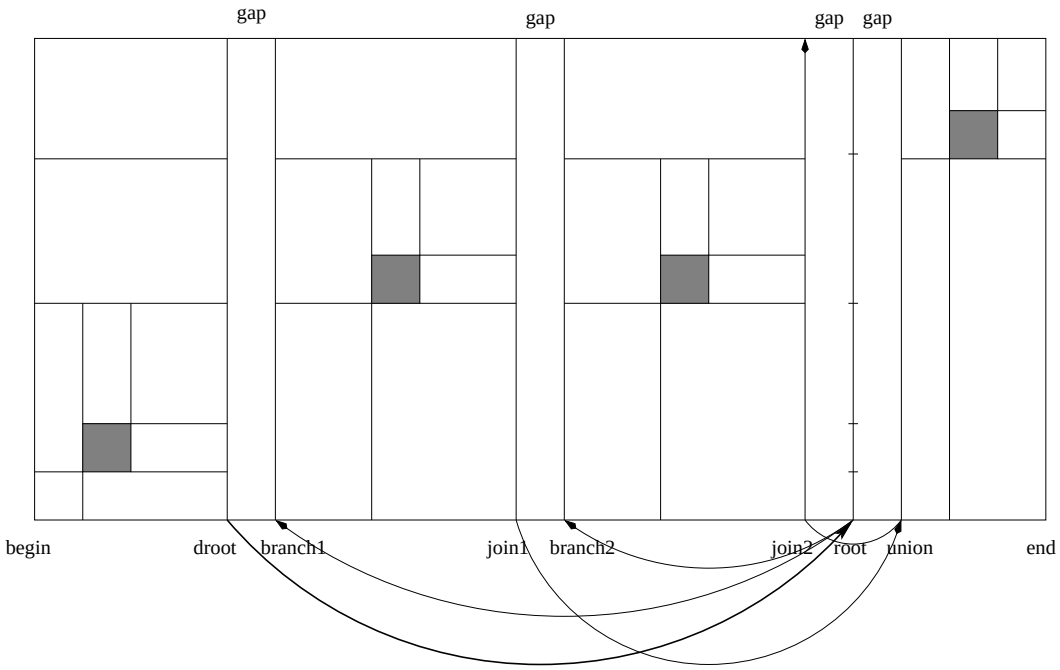

Figure 6.5. Two branch segments emerge from a single root.

several "blocks" of statements, with each block corresponding to a guard. These blocks are in sequence, so the gaps occur between the root of the branching structure (an "if" or "do" control point) and the guards, and there is also a gap after each block that transitions to the end of the branching structure.

The root of the structure is of special interest, and the SPIN parser gives it a sequence number greater than all of the branch segment blocks following guard control points. During the actual implentation of the partitioning, it was decided that the best way to treat this situation is to create degenerate, zero-area components at each root, and to partition them as one does the immediately preceding (geometrically) rectangular component that occurs before the branching structure begins. Doing so makes sense because of the geometric semantics of progress graph components, which attaches the left and upper boundaries of components to their rectangular interior.

6.3.3. Validation

6.3.3.1. Testing Numerical output was used for debugging at first; however, the partitioned progress graphs that were being produced quickly became complicated enough to warrant some effort toward producing graphical visual output automatically [26]. Later,

when working on the exploration of progress graphs, it was a huge benefit to be able to visualize execution paths through a state space.

6.3.3.2. Time Usage In order to be able to optimize the performance of the DRUiDD software, timing measuremants were implemented for many of its important functions. In the case of recursive functions, this task proved to be more challenging, but the problem was solved by stopping timers whenever certain function calls were made and starting them again immediately after the function call. In this way, even functions that called each other in a complex manner could be timed independently. These timing measurements facilitated verification of the correctness of the runtime performance data as well.

7. EXPERIMENT

7.1. MODELS

The models that were used for experimental measurements had from two to five processes each. Slightly different versions were used for SPIN and for DRUiDD model checkers, but they represented the same concurrent programs in every instance. A standard structure was used for the models, and then randomly selected processes were included into a model file from a pool of 45 processes that was generated for each parameter set – see Figure 7.1 and Figure 7.2 for examples.

The main parameters that were varied for the processes were: modulus of variables, length of statement blocks within branches, and sparseness of interactions among global variables and semaphores. The sparseness of interactions was controlled by increasing the total number of semaphores and global variables that were available for each process to access.

In each process, there was one outer do..od loop with up to three branches (one of which was an always-available option to break out of the loop). Also, within the loop, there was up to one conditional if..fi branching structure with up to three branches, as can be seen in Figure 7.3. 50% probability was set that any given statement would involve semaphores and there was a 50% probability that any given variable access was for a global variable (instead of a local one). Semaphore locks are accomplished by receipt of a token from a dedicated message queue for each semaphore, and unlocks are achieved when the token is returned to the queue. Since the queue has a buffer size of only one, the semaphore is binary.

7.2. PROCEDURE

A shell script using a program called ProcGen (that the author helped to write) was run to generate 45 processes for each parameter set that was used. Then, a shell script using ModGen created models and ran SPIN and DRUiDD.

Parameters for the models varied, but they were chosen so that the model checkers would be able to complete the verifications on the computers that were being used and so

```
#define p 0
chan sema[24] = [1] of {bit};
byte myg_0 = 7; byte myg_1 = 22;
byte myg_2 = 9; byte myg_3 = 19;
byte myg_4 = 17; byte myg_5 = 31;
byte myg_6 = 10; byte myg_7 = 12;
byte myg_8 = 9; byte myg_9 = 13;
byte myg_10 = 26; byte myg_11 = 43;
byte myg_12 = 2; byte myg_13 = 43;
byte myg_14 = 35; byte myg_15 = 22;
byte myg_16 = 12; byte myg_17 = 18;
byte myg_18 = 4; byte myg_19 = 40;
byte myg_20 = 11; byte myg_21 = 8;
byte myg_22 = 39; byte myg_23 = 45;

#include "../../processes3/processes348/processes34812/
processes348122/processSP_3_48_12_2_18.pr"
#include "../../processes3/processes348/processes34812/
processes348122/processSP_3_48_12_2_36.pr"
#include "../../processes3/processes348/processes34812/
processes348122/processSP_3_48_12_2_18.pr"
#include "../../processes3/processes348/processes34812/
processes348122/processSP_3_48_12_2_14.pr"

init
{
    sema[0]!p;    sema[1]!p;
    sema[2]!p;    sema[3]!p;
    sema[4]!p;    sema[5]!p;
    sema[6]!p;    sema[7]!p;
    sema[8]!p;    sema[9]!p;
    sema[10]!p;    sema[11]!p;
    sema[12]!p;    sema[13]!p;
    sema[14]!p;    sema[15]!p;
    sema[16]!p;    sema[17]!p;
    sema[18]!p;    sema[19]!p;
    sema[20]!p;    sema[21]!p;
    sema[22]!p;    sema[23]!p;
}
```

Figure 7.1. A sample SPIN model program.

that as many transitions would be taken as possible. Eventually, during the experimentation, only SPIN was run on the models at first, because that was enough to determine

```
#define p 0
chan sema[24] = [1] of {bit};
byte myg_0 = 7; byte myg_1 = 22;
byte myg_2 = 9; byte myg_3 = 19;
byte myg_4 = 17; byte myg_5 = 31;
byte myg_6 = 10; byte myg_7 = 12;
byte myg_8 = 9; byte myg_9 = 13;
byte myg_10 = 26; byte myg_11 = 43;
byte myg_12 = 2; byte myg_13 = 43;
byte myg_14 = 35; byte myg_15 = 22;
byte myg_16 = 12; byte myg_17 = 18;
byte myg_18 = 4; byte myg_19 = 40;
byte myg_20 = 11; byte myg_21 = 8;
byte myg_22 = 39; byte myg_23 = 45;
active proctype dijkstra0()
{
    sema[0]!p;}
active proctype dijkstra1()
{
    sema[1]!p;}

/* dijkstra2() through dijkstra22() deleted
to conserve space on this page */

active proctype dijkstra23()
{
    sema[23]!p;}

#include "../../processes3/processes348/processes34812/
processes348122/processDR_3_48_12_2_18.pr"
#include "../../processes3/processes348/processes34812/
processes348122/processDR_3_48_12_2_36.pr"
#include "../../processes3/processes348/processes34812/
processes348122/processDR_3_48_12_2_18.pr"
#include "../../processes3/processes348/processes34812/
processes348122/processDR_3_48_12_2_14.pr"
```

Figure 7.2. A sample DRUiDD model program.

whether the value of the independent variable was in a desirable range for data collection: it would have been pointless to run the same types of models repeatedly.

Because large models are sometimes difficult to model check completely, there were occasions when a verification did not finish successfully. From time to time, when a verification seemed to be stalled, it was terminated by the experimenter. Whenever one

```
active proctype user_3_48_12_2_14()
{
  bit mySkip;
  byte local_0 = 43; byte local_1 = 20; byte local_2 = 39;
  byte local_3 = 42; byte local_4 = 1; byte local_5 = 20;
  byte local_6 = 23; byte local_7 = 24; byte local_8 = 39;
  byte local_9 = 10; byte local_10 = 18; byte local_11 = 16;
  mySkip = 0;
  do  :: (1) ->    break;
      :: ((myg_17) % 48==(local_8) % 48) ->sema[1]?p;
        myg_6 = (myg_22) % 48; sema[1]!p;
        mySkip = 0;
        if  :: (1) ->    mySkip = 0;
            :: ((myg_20) % 48!=(local_5) % 48) ->
              local_11 = (myg_11) % 48;
              myg_11 = (local_10+myg_15+myg_4) % 48;
              sema[18]?p; sema[3]?p;
              myg_7 = (myg_21+local_9) % 48; sema[3]!p;
              local_9 = (local_2) % 48; sema[18]!p;
              sema[21]?p; myg_5 = (myg_0) % 48;
              sema[21]!p; myg_16 = (local_3) % 48;
              mySkip = 0;
        fi;
        local_11 = (myg_9) % 48; sema[12]?p;
        myg_3 = (myg_18) % 48; sema[15]?p;
        myg_4 = (myg_7+myg_2+local_11) % 48;
        sema[2]?p; sema[20]?p;
        local_5 = (local_10+local_11+local_10) % 48;
        local_7 = (local_8+local_0) % 48;
        sema[2]!p; sema[12]!p; sema[15]!p;
        sema[20]!p;
        mySkip = 0;
  od;
}
```

Figure 7.3. A sample process used in the experiments.

or both of the model checkers being compared failed to complete its verification due to the depth-limit or memory limitations of the computer being used, the data point was discarded.

The depth-limit that was used was increased over the course of the experiment from 10,000 to 100,000 to 200,000 to 5,000,000 as it became apparent that the asymptotic behavior could best be derived from doing so.

Model parameters were changed to explore the asymptotic behavior as thoroughly as possible during the time available for experimentation. In Table 7.1, a section of the data for small models (with few transitions) is displayed.

Table 7.1. Data was sorted according to the number of transitions that SPIN took to verify each model.

model	depth	error?	SPIN states	matched	transitions	depth	error?	DRUiDD states	matched	transitions	disagree?
23161227	180	31	0	58	24	82	272	23	0	48	9	57	0
23241227	180	31	0	58	24	82	272	23	0	48	9	57	0
23481227	276	31	0	58	24	82	464	23	0	48	9	57	0
23161623	232	39	0	74	32	106	356	31	0	56	9	65	0
23161627	236	39	0	74	32	106	360	34	0	55	8	63	0
23163213	240	39	0	74	32	106	364	31	0	56	9	65	0
23163217	240	39	0	74	32	106	364	34	0	55	8	63	0
23241623	232	39	0	74	32	106	356	31	0	56	9	65	0
23243213	240	39	0	74	32	106	364	31	0	56	9	65	0
23481623	372	39	0	74	32	106	620	31	0	56	9	65	0
23481627	372	39	0	74	32	106	620	34	0	55	8	63	0
33641627	404	42	0	78	32	110	652	31	0	140	49	189	0
33481627	404	42	0	78	32	110	652	31	0	140	49	189	0
23481234	380	40	0	78	36	114	664	35	0	40	1	41	0
23161231	268	43	0	81	36	117	408	35	0	60	9	69	0
23161236	260	43	0	81	36	117	400	35	0	60	9	69	0
23241231	268	43	0	81	36	117	408	35	0	60	9	69	0
23241236	260	43	0	81	36	117	400	35	0	60	9	69	0
23481231	412	43	0	81	36	117	696	35	0	60	9	69	0
23481236	404	43	0	81	36	117	688	35	0	60	9	69	0
333212110	164	19	0	88	86	174	252	14	0	35	8	43	0
23161223	180	31	0	109	97	206	272	26	0	47	8	55	0
23241223	180	31	0	109	97	206	272	26	0	47	8	55	0
23241226	180	31	0	109	97	206	272	23	0	48	9	57	0
23481223	276	31	0	109	97	206	464	26	0	47	8	55	0
23481226	276	31	0	109	97	206	464	23	0	48	9	57	0
33481226	300	34	0	115	97	212	488	23	0	132	49	181	0
33641624	380	39	0	141	129	270	628	31	0	56	9	65	0
33481624	380	39	0	141	129	270	628	31	0	56	9	65	0
23161624	240	39	0	141	129	270	364	31	0	56	9	65	0
23161628	240	39	0	141	129	270	364	31	0	56	9	65	0
23163214	240	39	0	141	129	270	364	34	0	55	8	63	0
23163218	244	39	0	141	129	270	368	31	0	56	9	65	0

It was decided that the number of transitions taken by SPIN for verification of a model was the best measure of the complexity of the models in the testbed, so this became the independent variable for the results. Dependent variables were ratios of DRUiDD's performance over SPIN's performance. DRUiDD preprocessing time/PAN total time was the first dependent variable. The second dependent variable was DRUiDD search time/PAN total time, and the third was DRUiDD total time/PAN total time. The last dependent variable was the number of transitions for DRUiDD divided by the number of transitions for SPIN. The data was partitioned into two sets depending on whether deadlock was detected or not.

8. PERFORMANCE

8.1. COMPONENT SEARCH

One of the primary results about component search was that the number of states could be reduced. Figure 8.1 shows that the inductive method outperforms SPIN and that the recursive method outperforms the inductive method asymptotically (longer processes with more semaphores).

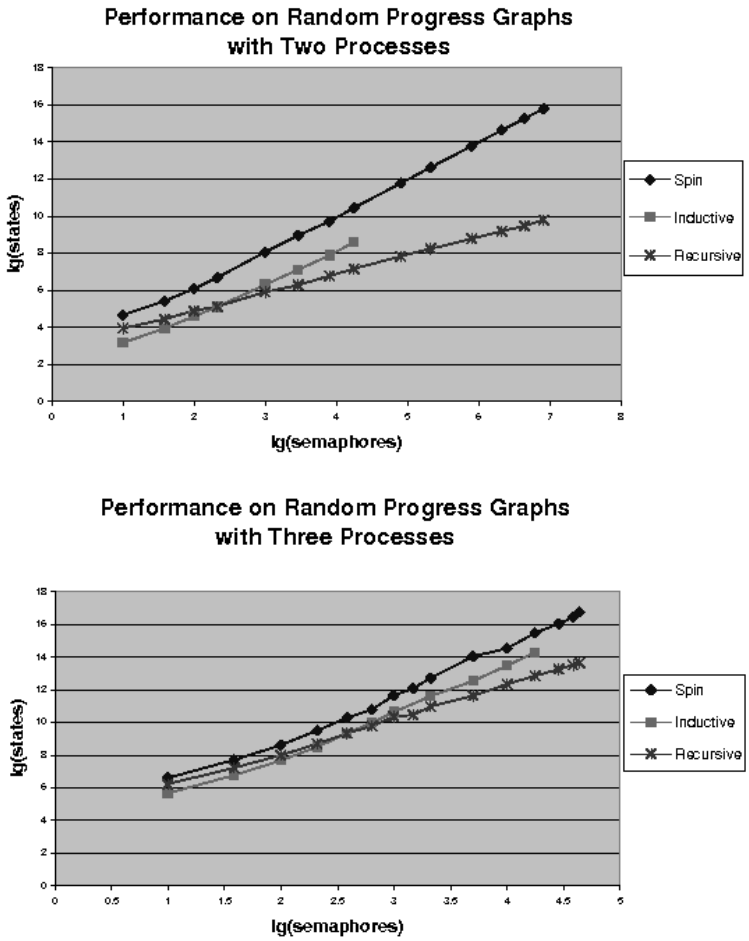

Figure 8.1. Comparison of SPIN to the inductive and recursive methods (on two-process and three-process models) in terms of the number of states for SPIN and the number of components for the others.

8.2. STATE SEARCH

In the following graphs, each data point represents the measured values for a particular randomly generated model having a small number of processes. The number of transitions taken by SPIN to verify the model is the x-coordinate of each data point, and the y-coordinate is a ratio of some measured value for DRUiDD verification divided by a relevant measurement for SPIN on the same model using the same equipment to do the measuring. The reason for choosing the number of transitions for SPIN as the independent variable is that it is a good indication of the difficulty of verifying deadlock-freedom for a model. Asymptotic behavior is noted for each result.

From Figure 8.2, the ratio of the number of transitions for SPIN and the number of transitions for DRUiDD – when the models are deadlock-free – is below the value of one after the value of ten million on the x-axis. This value of ten million is actually the number of transitions for SPIN, so this implies that one can expect that DRUiDD will use fewer transitions than SPIN whenever SPIN uses more than ten million.

The same conclusion may be drawn in the case when deadlock occurs as when it does not occur, but the point at which the asymptotic behavior is reached (the y-value of one) is closer to two million than to ten million. This is illustrated in Figure 8.4.

In terms of runtime, the time was measured for DRUiDD for both the preprocessing phase and the search phase, which together make up the total runtime. Graphs for the deadlock-free models (shown in Figure 8.6) indicate that the ratio of the preprocessing time (over the total time taken by SPIN) shrinks quickly.

As with case that was just discussed, when deadlock does occur, the runtime ratio for preprocessing by DRUiDD over the total runtime for SPIN shrinks very quickly, as seen in Figure 8.8.

Total runtime for DRUiDD divided by total runtime for SPIN is the litmus test for efficiency, and the graphs of this data are collected in Figure 8.10 and Figure 8.12. DRUiDD performs well when SPIN uses many transitions for a verification, and it is especially fast when, in addition, there is a possibility of deadlock in the program.

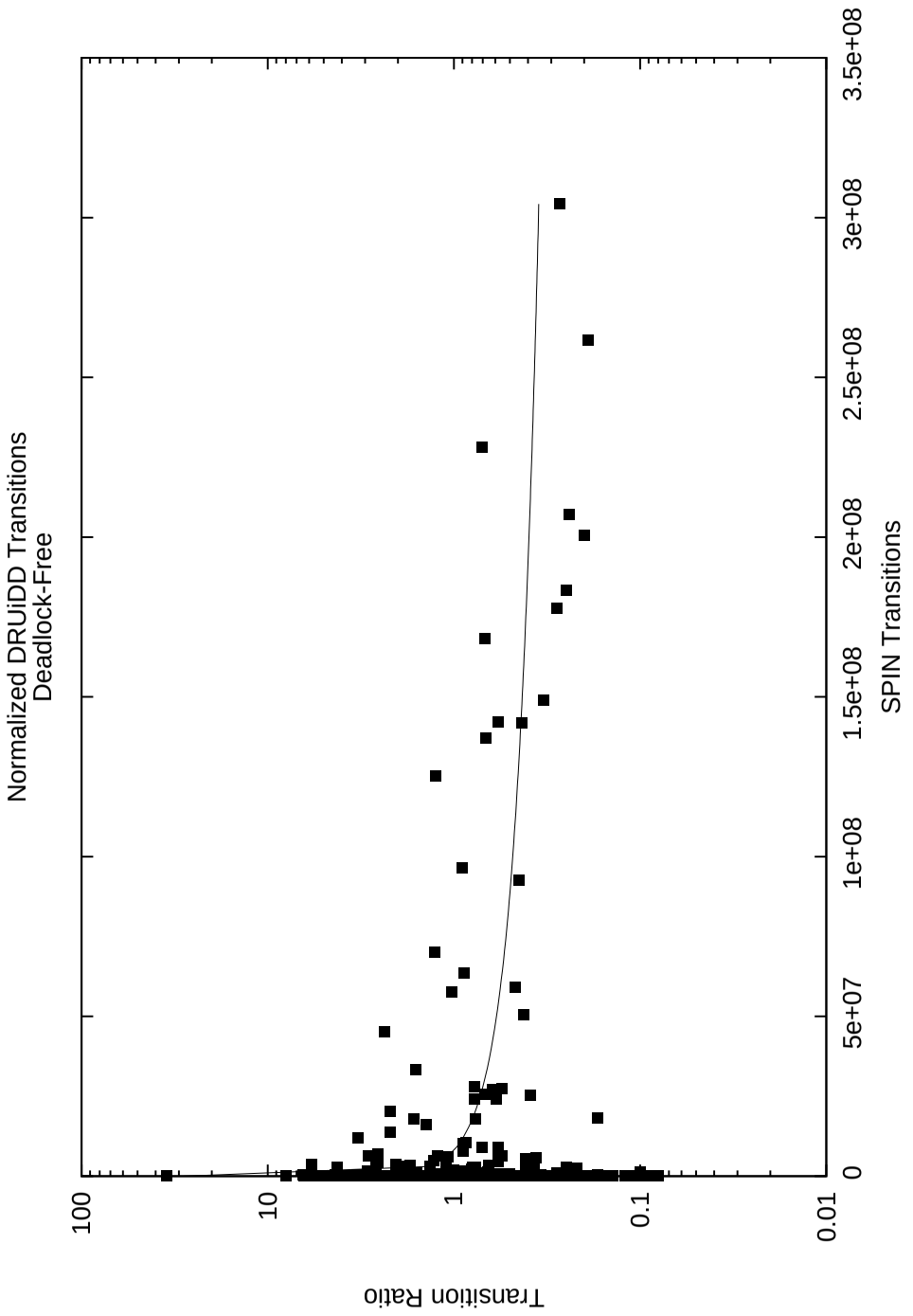

Figure 8.2. The Dihomotopic Progress Shell Decomposition technique reduces the number of transitions that searches took in verifications where deadlock was not found.

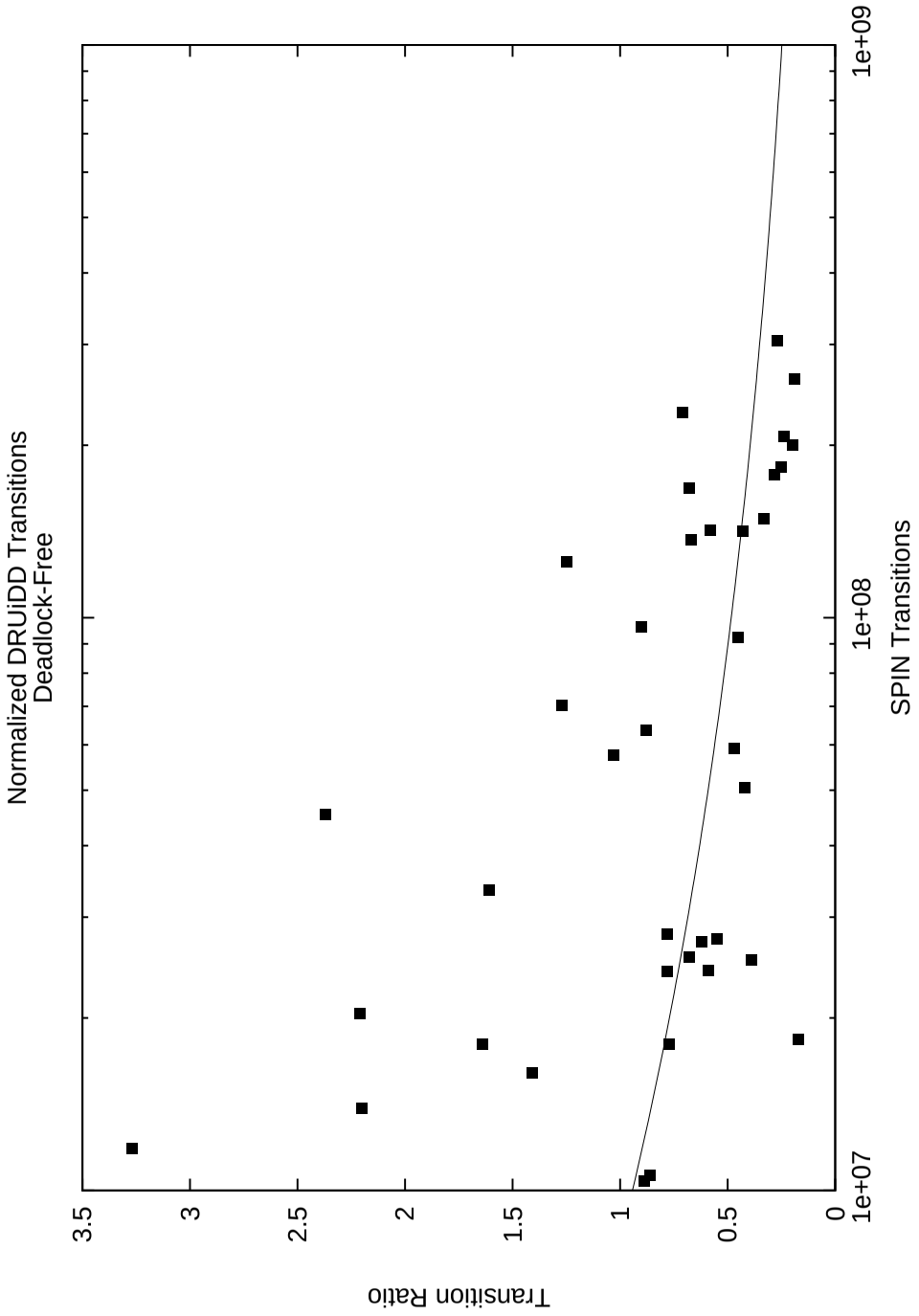

Figure 8.3. Asymptotic behavior of Figure 8.2 is seen by zooming in on the far right tail of the graph. The scale of the axes has been reversed as well, with logarithmic scale on the horizontal axis.

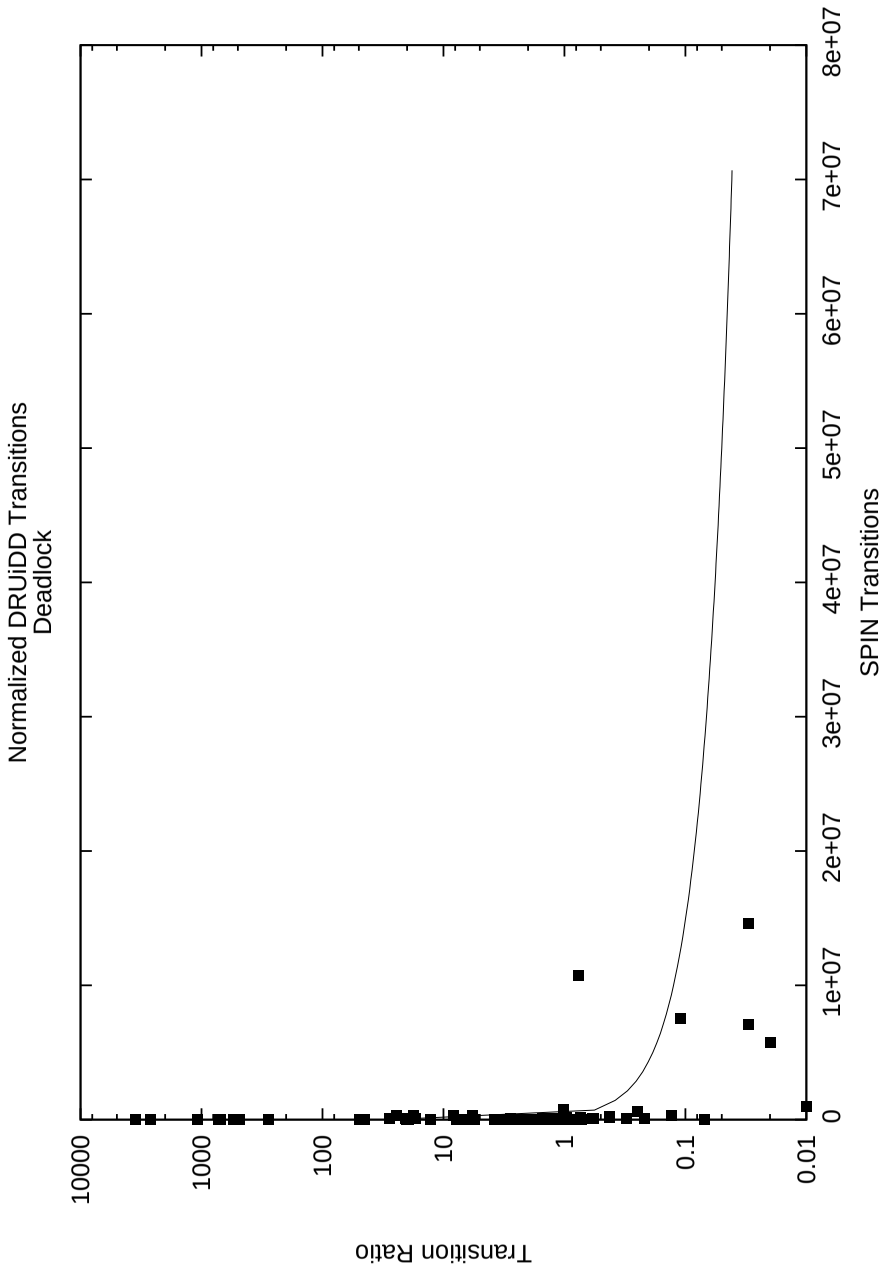

Figure 8.4. The Dihomotopic Progress Shell Decomposition technique reduces the number of transitions that searches took in verifications where deadlock was found.

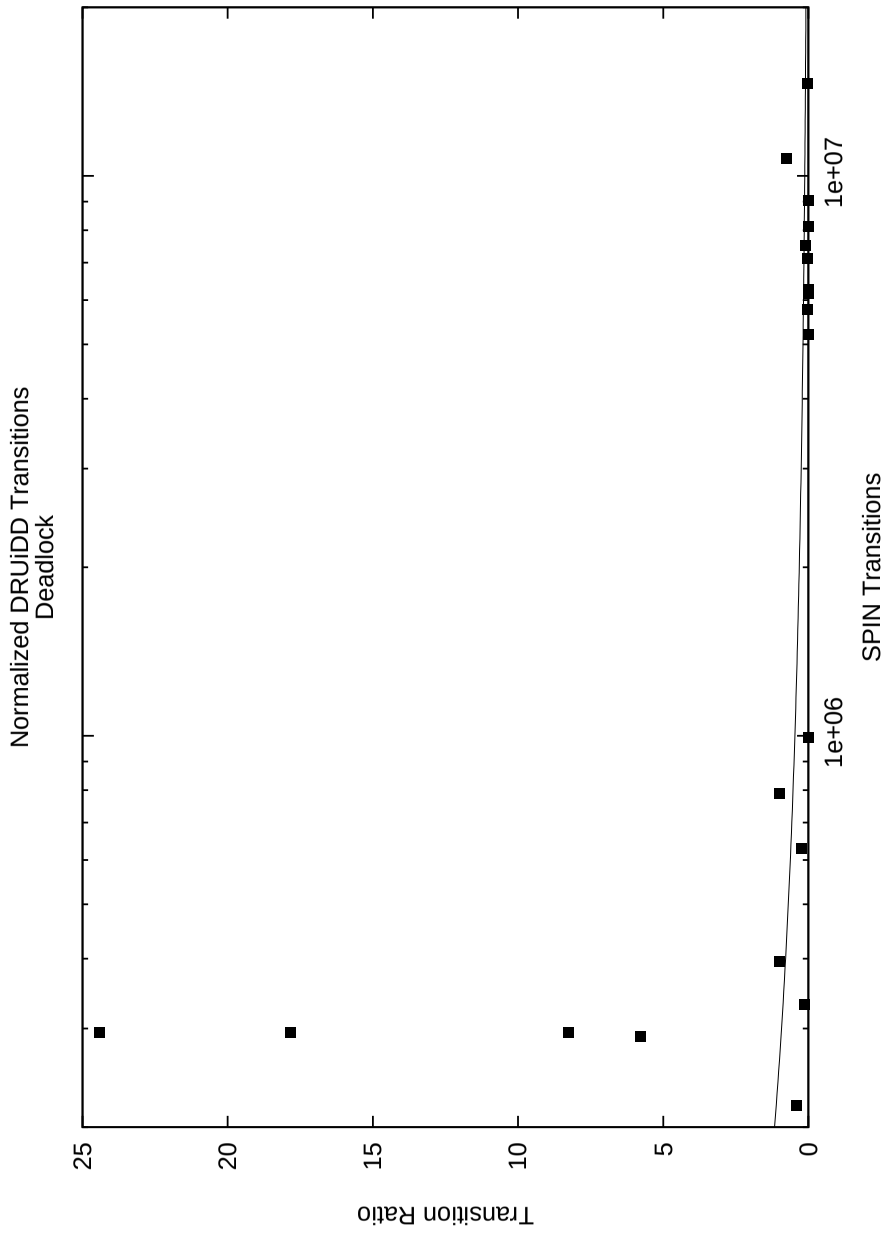

Figure 8.5. Asymptotic behavior of Figure 8.4.

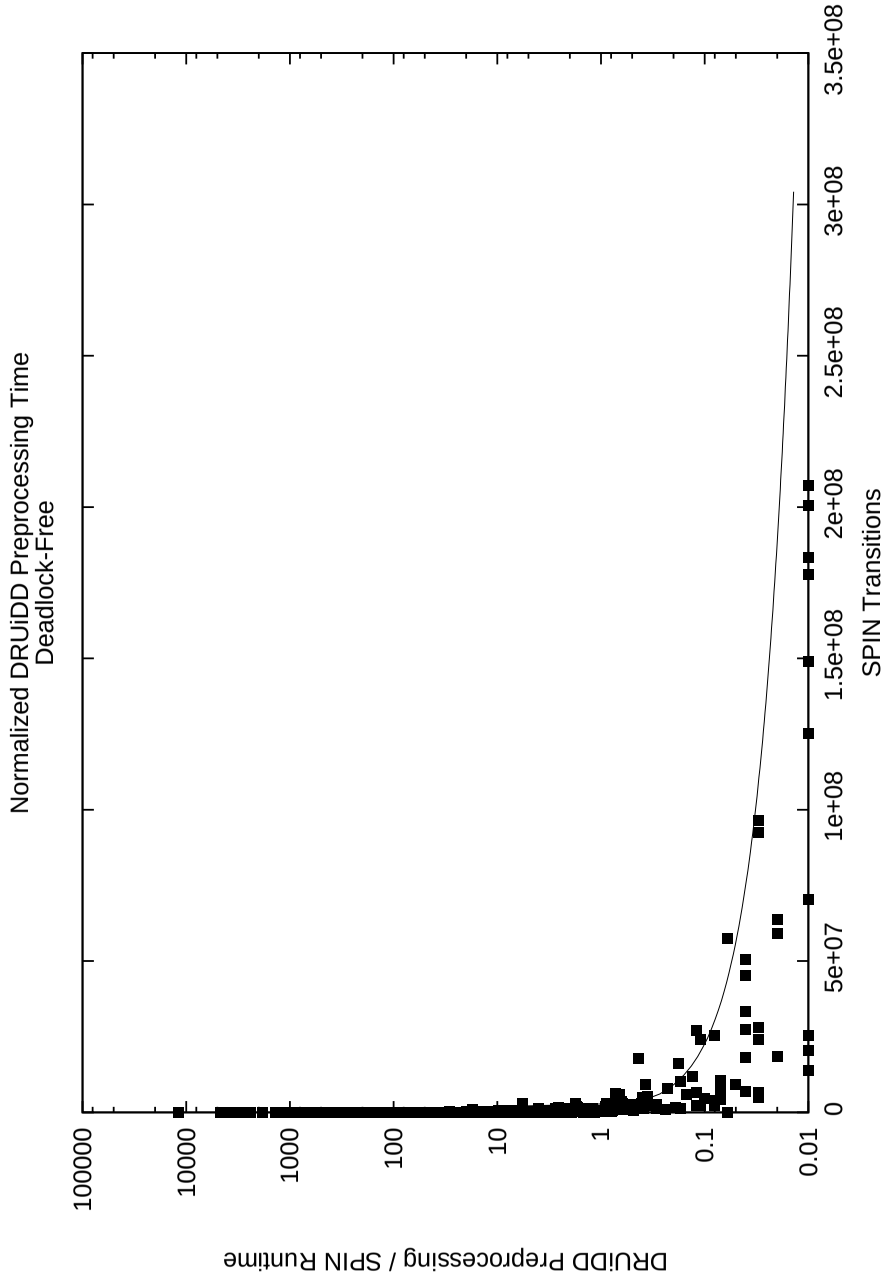

Figure 8.6. Ratio of DRUiDD preprocessing time to total time for SPIN when deadlock was not present.

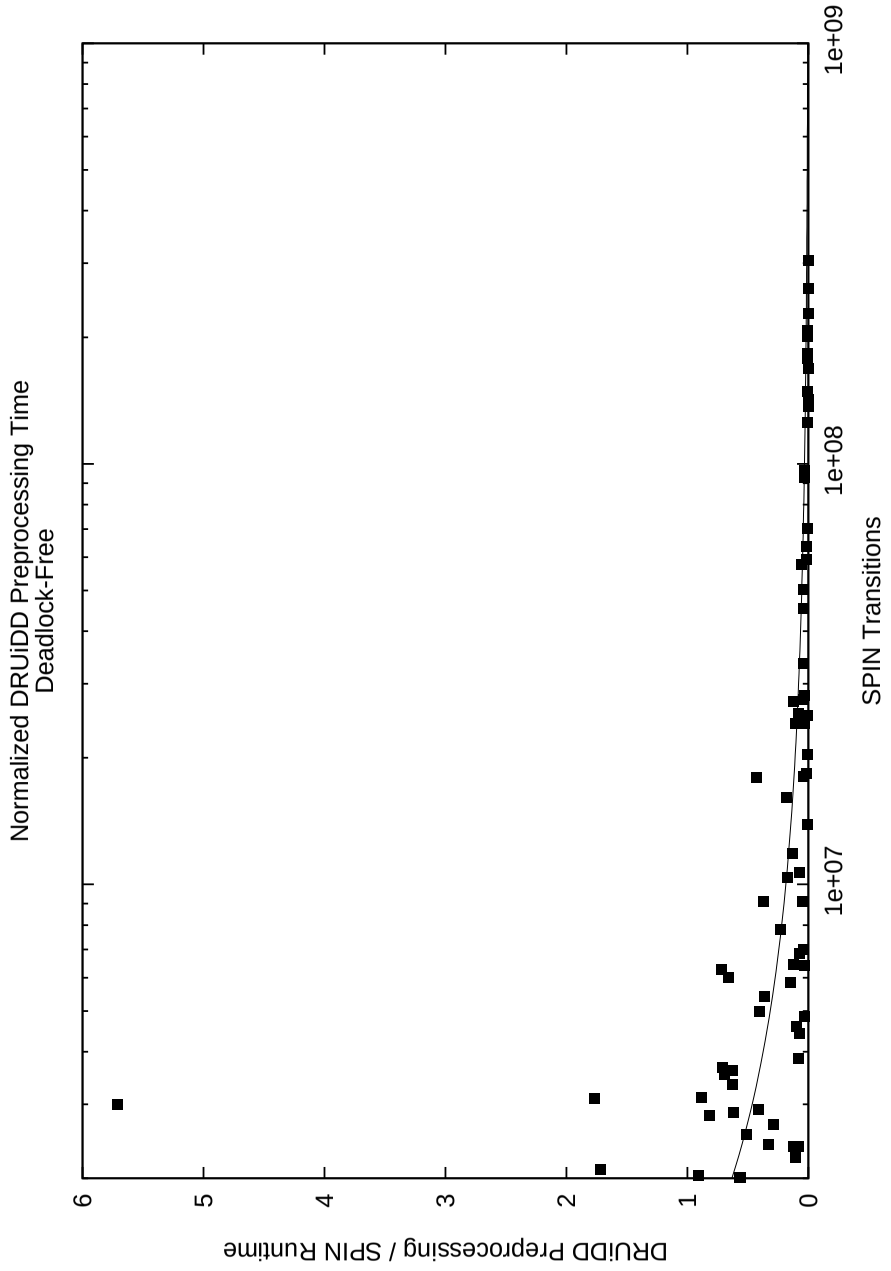

Figure 8.7. Asymptotic behavior of Figure 8.6.

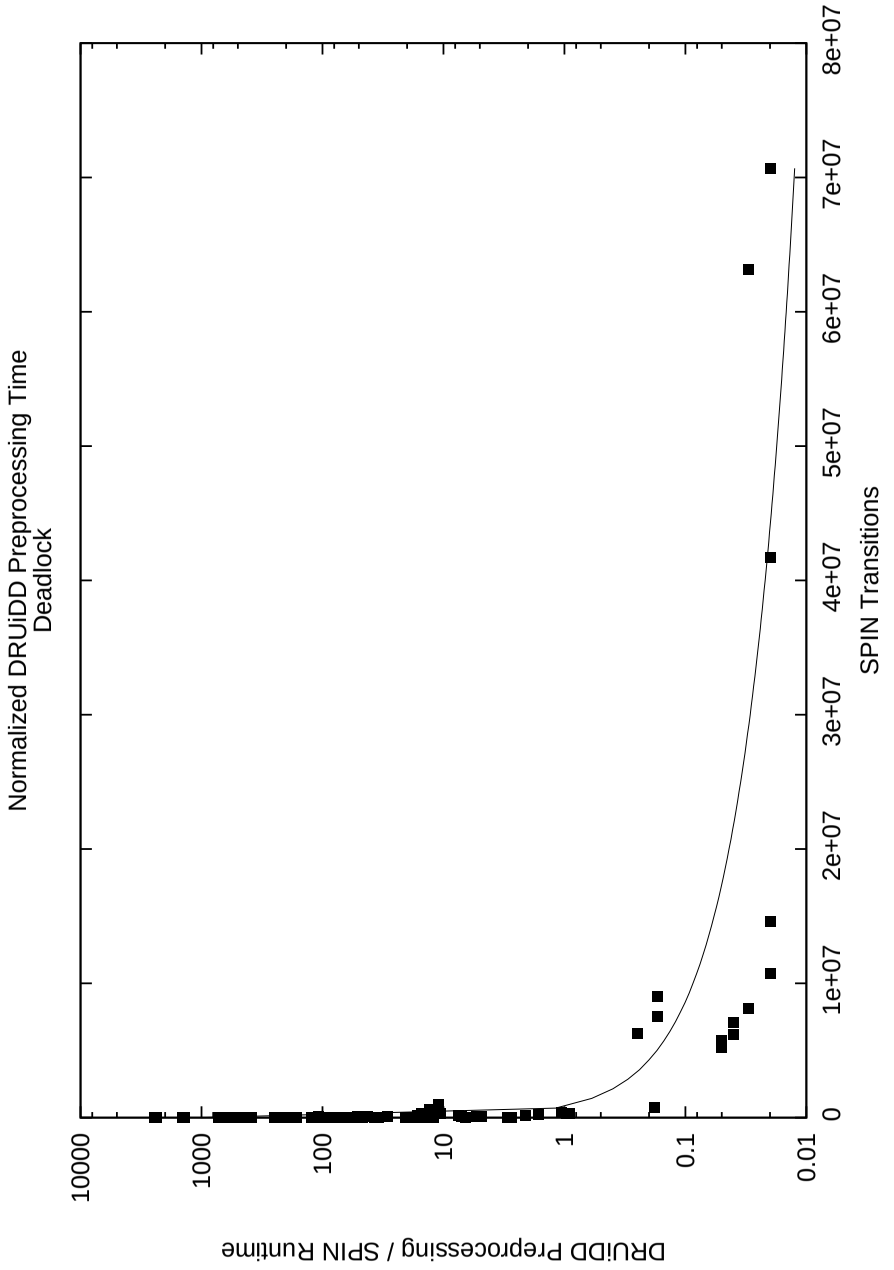

Figure 8.8. Ratio of DRUiDD preprocessing time to total time for SPIN when deadlock was discovered.

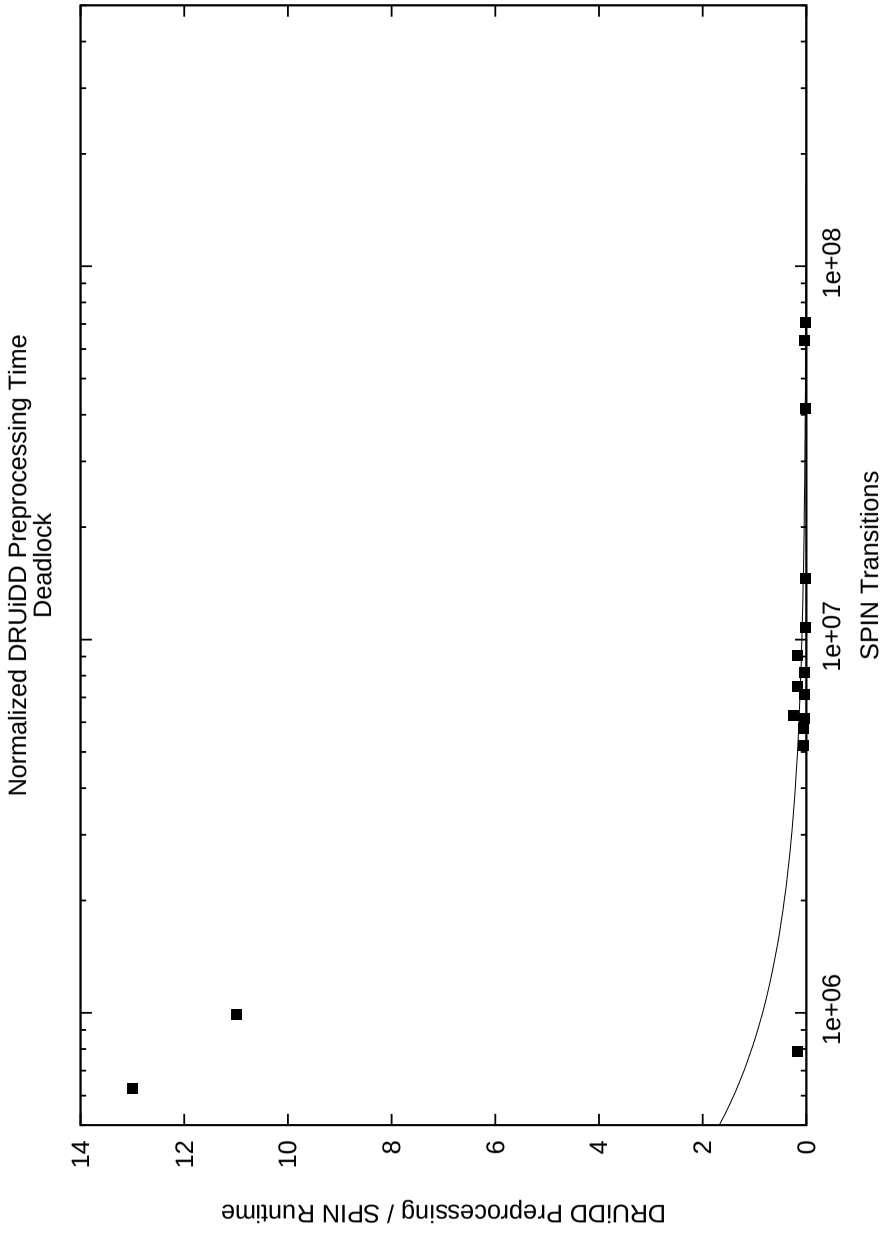

Figure 8.9. Asymptotic behavior of Figure 8.8.

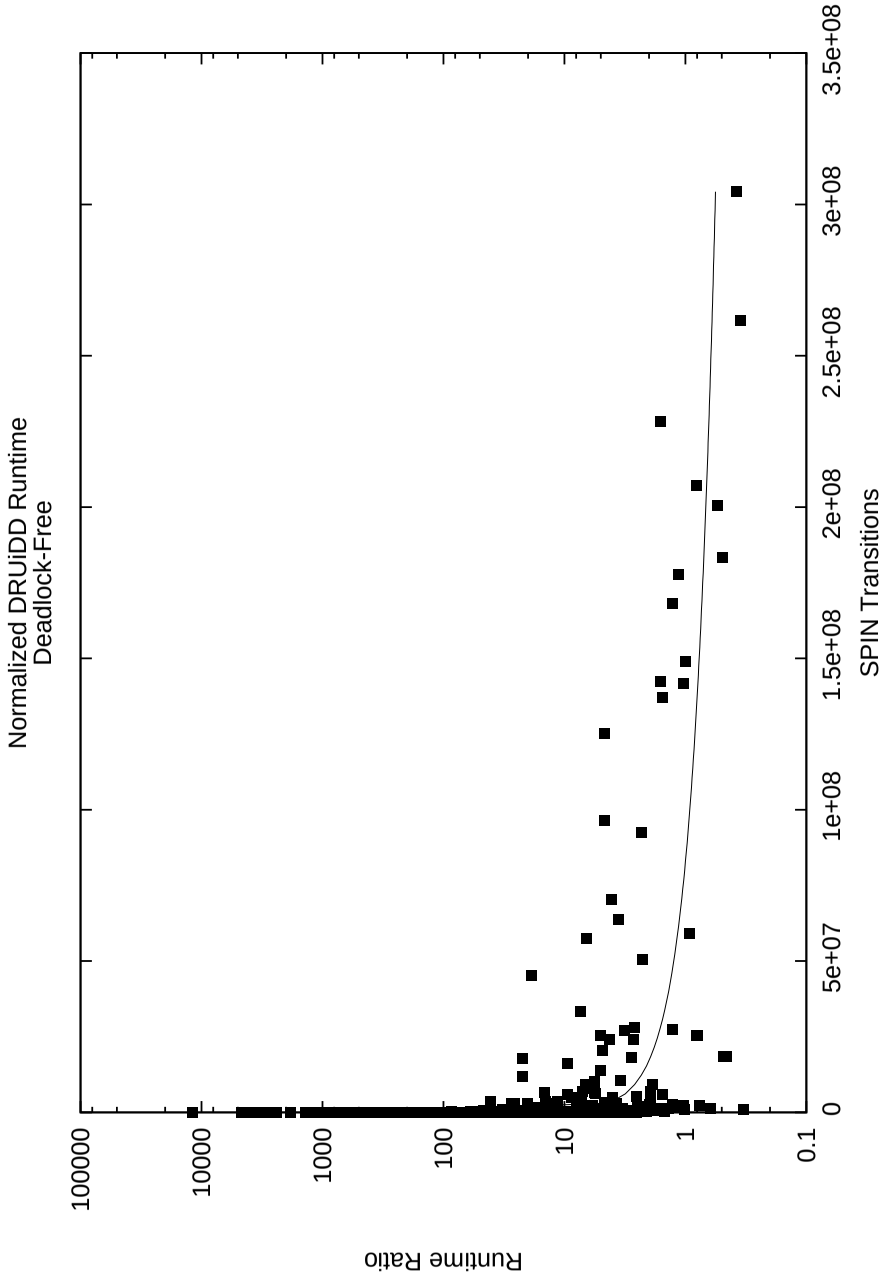

Figure 8.10. Ratio of DRUiDD total runtime to total runtime for SPIN when deadlock was not present.

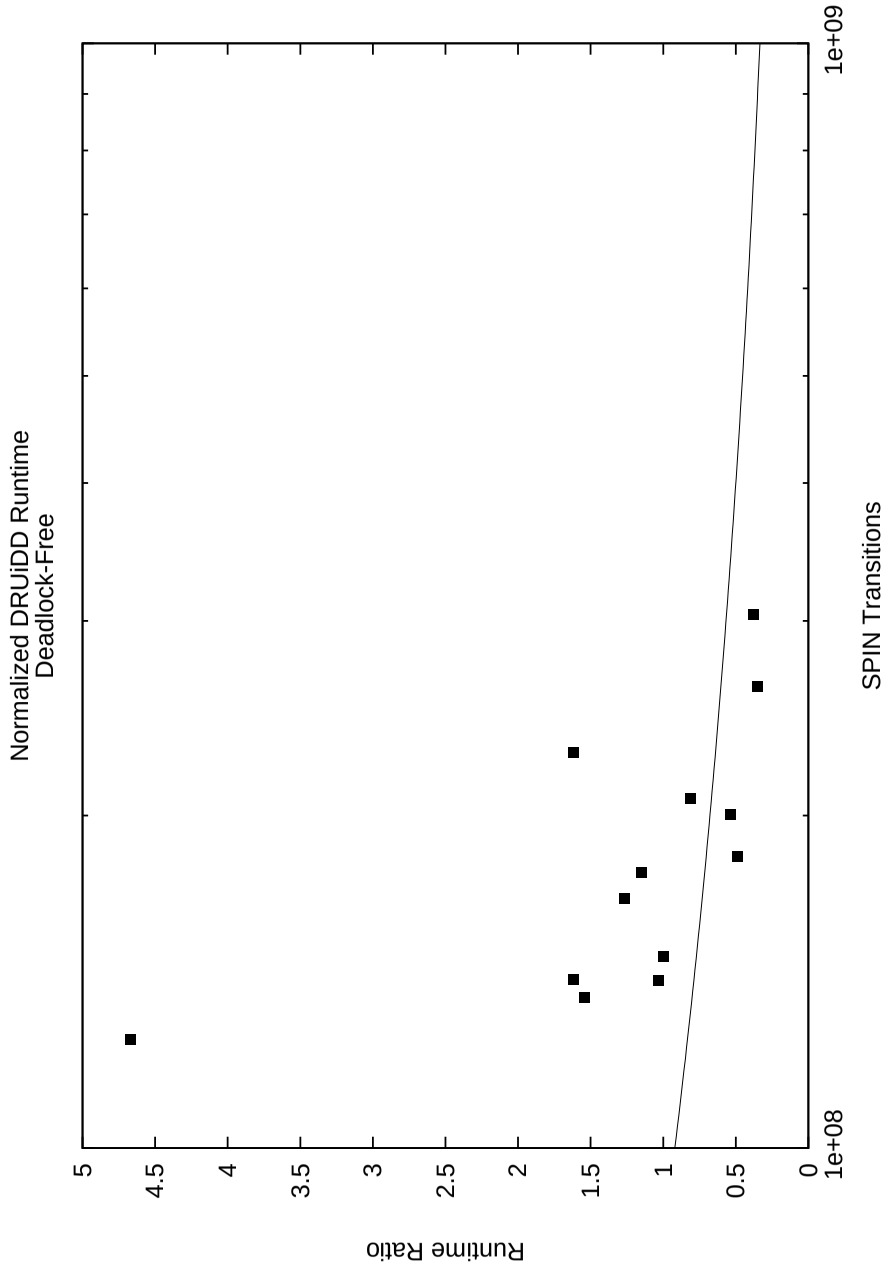

Figure 8.11. Asymptotic behavior of Figure 8.10.

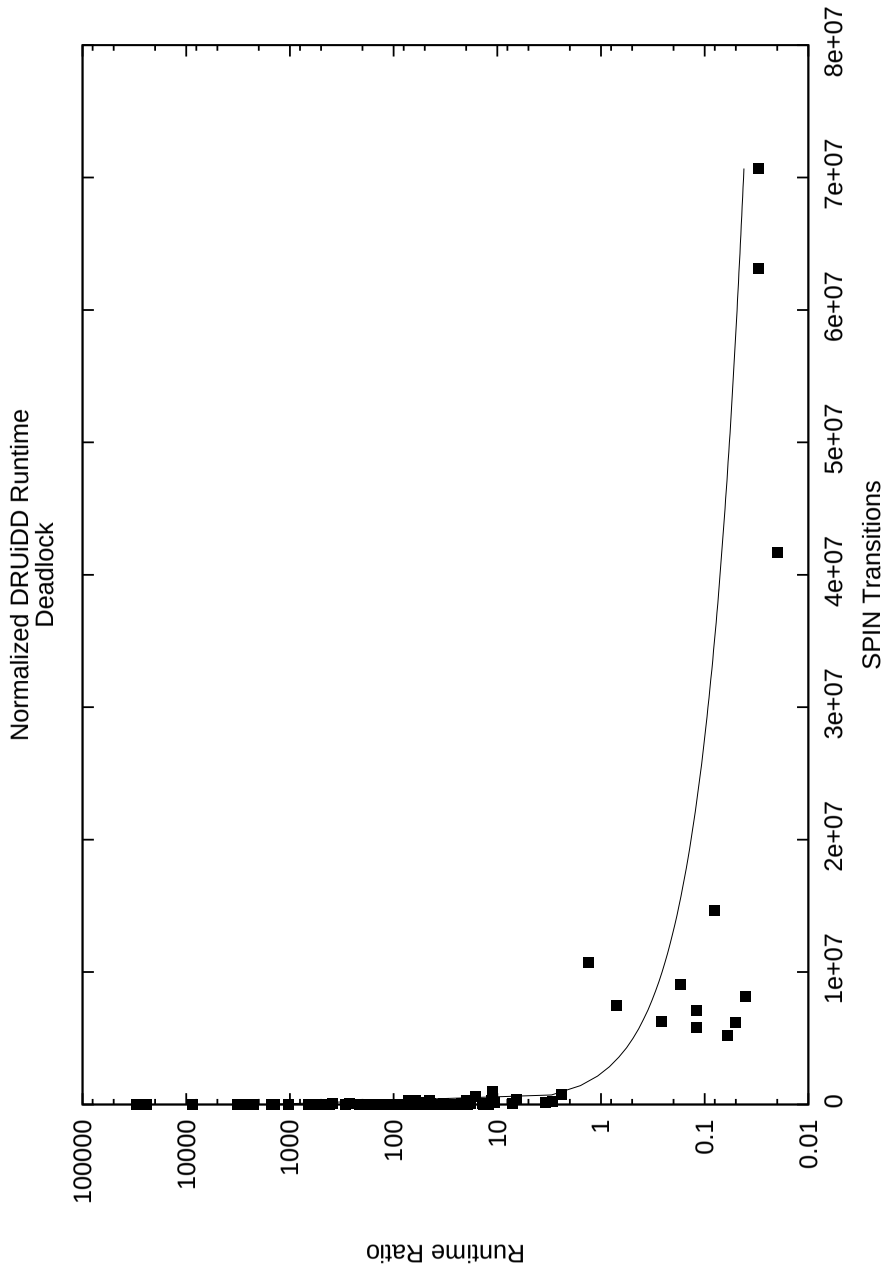

Figure 8.12. Ratio of DRUiDD total runtime to total runtime for SPIN when deadlock was found.

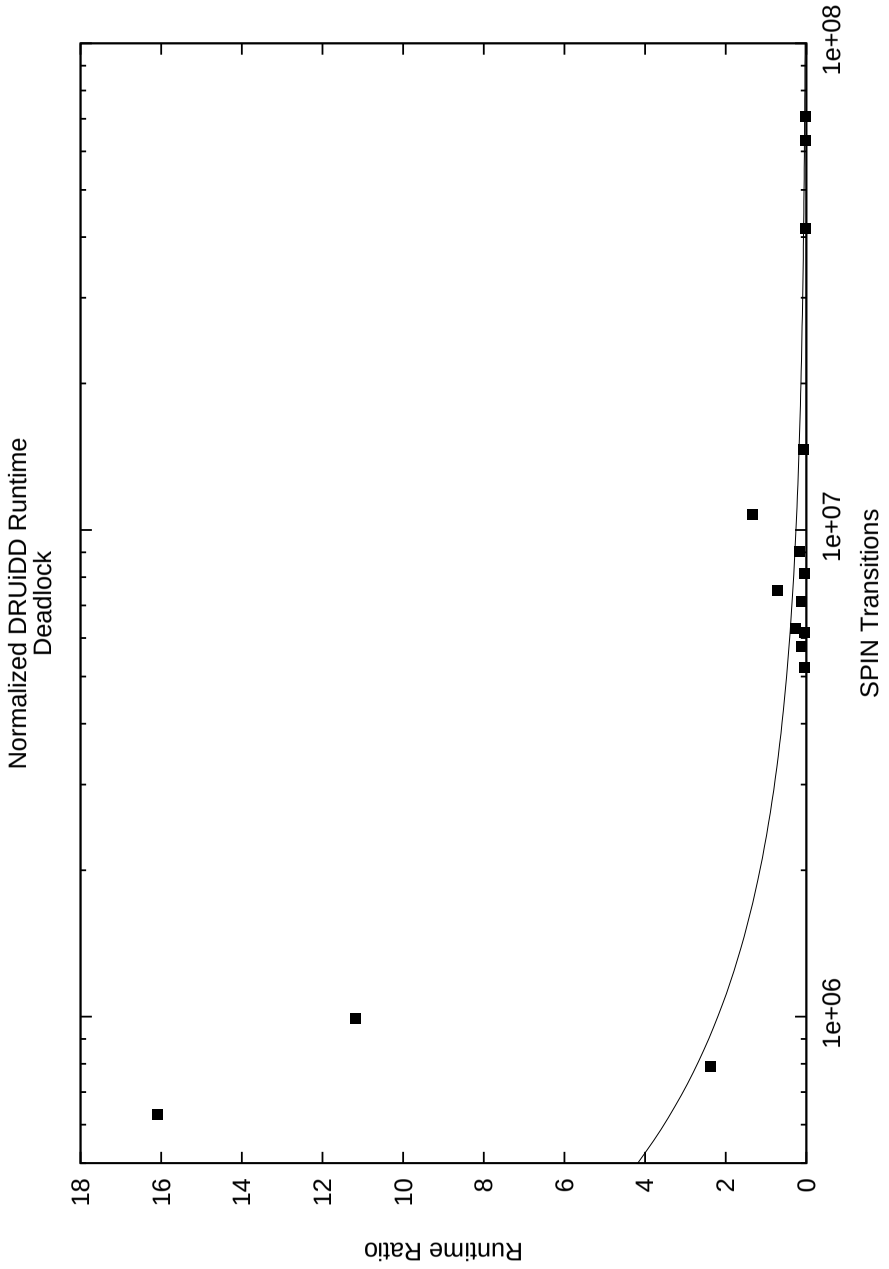

Figure 8.13. Asymptotic behavior of Figure 8.12.

9. FUTURE WORK

9.1. INTEGRATING PARTIAL-ORDER REDUCTION

Among currently accepted general model checking optimization techniques, partial-order reduction has proven itself to be perhaps the most important. Therefore, it would be a great boon to the geometric technique of dihomotopic reduction if partial-order reduction is compatible with it, as claimed by Goubault and Raussen [9].

9.2. SEARCH OPTIMIZATION

Two other potential optimization techniques are discussed here. The first of these involves a method for recognizing states that represent possible early termination points for the search path. The second technique uses a strategy for exploring increasingly extensive subsets of the state space.

9.2.1. Future State Projection Depth-first search works by successively searching as far as possible into a state space without going through states that have been already visited. The objective of DRUiDD is to find a reduced state space for exploration that is still sufficiently large for deadlock to be detected accurately. If deadlock is possible in a progress graph, then, by definition, there is a point of deadlock that is reachable from the origin. Now, if the search has already progressed to a certain state, one can consider the set of states that are reachable from it. Each transition that is taken potentially reduces the set of reachable states from the current state. What could be useful is to calculate and store the set of states that are reachable within the current component as soon as it is entered from a neighboring component. Because it is only necessary to search rays parallel to the axes extending forward from the point of entry into the component, it is never necessary to enter the region of the component that is in the future of an entry point later in the search. Thus, one can consider those points to be already visited and terminate search paths that later visit them.

9.2.2. Iterative Thickening Consider the component boundaries of the partition that is used in DRUiDD. Those components are N-dimensional cuboids, and their boundaries are composed of cuboids of dimension zero up through $N-1$. Let M be an index

that runs from one to $N-1$. There is an increasing nested sequence of cuboids as one takes the union $U_M(C)$ of the cuboids (of dimension less than or equal to M) that are on the boundary of each component C. Prior to each new component C_1 being entered, a state is visited that is a point of exit from the previous component C_0. If that state is within $U_M(C_0)$, then allow any transition that is parallel to a boundary component of $U_M(C_0)$. In this way, as M increases, more of the state space is searched, and when M equals $N-1$, all of the state space is searched that the original DRUiDD would have searched.

9.3. MORE EXPRESSIVE MODELS

The version of DRUiDD that has been described and tested is capable of verifying deadlock-freedom for a restricted class PROMELA models. This class contains models that can express the most important features of a concurrent program, but not some other features that the more general software SPIN is able to process. Function calls and dynamic process spawning are permitted in models for input to SPIN but not to DRUiDD. Another difference is that DRUiDD does not process models that do any message-passing other than basic locking and unlocking of semaphores via the exchange of a token through message channels with a buffer of size equal to one. In general, one would want to be able to use channels with larger buffers that could also accept messages with more interesting data. Allowing rendez-vous communication would significantly expand the expressiveness of the class of models for DRUiDD as well.

9.4. TRACE-CONSISTENT LTL

Whereas deadlock-freedom is probably the easiest interesting property for a model checker to verify, there are other safety properties and also liveness properties that could be processed by an extended version of DRUiDD. That is, any trace-consistent property should be verifiable using the dihomotopic reduction technique, where trace-consistent means here that the verification outcome is the same on any execution path that represents a given trace. The automata-theoretic foundation for such an extension has already been laid by Bollig and Leuker [20].

10. CONCLUSION

The purpose of this study was the investigation of the potential for more efficient deadlock detection by using a geometric understanding of state spaces of concurrent programs. Doing so required development of a model checker that is capable of parsing and checking basic models similar to the ones that SPIN does. The software program DRUiDD is based on SPIN structurally, but it uses a preprocessing phase that analyzes the geometry of the model that is presented for verification. Deadlock detection is the sole aim of the current version of DRUiDD. The results of this study will motivate further development of the technique of dihomotopic reduction.

Those results strongly suggest that the preprocessing phase can be done efficiently enough to make it worthwhile for large model verifications. In fact, there is an observable trend in the data that substantiates the hypothesis that dihomotopic reduction outperforms partial-order reduction asymptotically and that the cost of the preprocessing becomes negligible. There is indeed margin for improvement in the DRUiDD software design and implementation that could provide even stronger evidence in support of that hypothesis. Moreover, when deadlock is found for large models, the efficiency gained is even greater asymptotically than when it is not present.

Deadlock detection is the most basic of properties that model checkers can attempt, yet it is representative of the issues of safety and also reachability. Other work has shown that for trace-consistent linear-time temporal formulas, there is a corresponding class of automata. Therefore, dihomotopic reduction should not be limited to use in deadlock detection alone. To be fair, there is a certain amount of mathematical preparation needed to assimilate the foundational topological background for understanding dihomotopy, but that investment, like the preprocessing phase of DRUiDD itself, has demonstrated its value in this dissertation project.

APPENDIX

```
/* FILENAME:  dr-main.h */

#ifndef DRUIDD_H
#define DRUIDD_H

#define BRANCHES 4
#define  FORKS   5
#define  SEGS   1024
#define ADJACENCY_SIZE 2048

/* Constants */
const short LINE_TYPE_BLACK = 1;
const short LINE_TYPE_BLUE = 2;
const short LINE_TYPE_GREEN = 4;
const short LINE_TYPE_PINK = 8;
const short LINE_HORZ = 1;
const short LINE_VERT = 2;
const short LINE_FORWARD = 1;
const short LINE_BACKWARD = 2;


char *emalloc(unsigned long);

void * dacmalloc(unsigned long n)
{
  return (void *) emalloc(n);
}

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pan.h"
#include "dr-constants.h"
#include "dr-types.h"
#include "dr-data.h"
#include "dr-time.h"
#include "dr-general.h"
#include "dr-visualize.h"
#include "dr-partitioninit.h"
#include "dr-partitionsec.h"
#include "dr-graph.h"
#include "dr-analyze.h"
#include "dr-search.h"

/* April 20, 2009 */
```

```
/* Created by David A. Cape (dacvdc@mst.edu) in the */
/* Computer Science Dept. of Missouri S&T. */
/* Intended for use in modifying SPIN source code for */
/* dissertation research with Prof. Bruce M. McMillin. */
/* Expanded significantly and completed */
/* by Stephen Jackson (scj7t4@mst.edu). */

/* Helpful Notes: */
/* - Arguments i & j always refer to a pair of processes, */
/*   where i is the process on the x axis */
/*   and j is the process on the y axis */
/* - General Functions are intended to be used throughout */
/*   the code, not simply when one function is */
/*   needs a specific feature */
/* - Helper Functions are typically added by SCJ and are */
/*   outside of the original specfication of DRUiDD */
/*   often they are used to attempt to organize the many */
/*   steps some of the processing steps need */
/* - Processing Functions are noted by the stage they are */
/*   used in, divided into to 3 Categories: */
/*   Pre-Partitioning, Partitioning, and Post-Partitioning */
/* - Call denotes what the function typically calls the */
/*   one listed. Various implies that the function could */
/*   be used throughout DRUiDD as needed to correct issues */
/*   within the data / general computation */
/* - Apology: Due to a lack of foresight, the word segment */
/*   is used interchangibly to refer to 2 things: */
/*   the transition number of a critical change, and a */
/*   branch of a control structure */
/* - Functions whose purpose begins with [DEBUG] are just */
/*   that: Debug functions. They make no changes to */
/*   Any structure. It is safe to remove them for the */
/*   purpose of speed gains. Most of these are also */
/*   Given a debug_ prefix in the function name */
/* - Add line functions are recursive and extend lines all */
/*   over the place. (REMOVED 8-3-09) */
/* - Some of David's Documents and old versions of this */
/*   code may mention or use the line method for secondary */
/*   Partitioning & Decomposition. This is no longer the */
/*   case. This is now done using rectangluar regions */
/*   Breaking up other rectanglar regions based on their */
/*   respective edges. Improvements might be made on */
/*   Case detection for rectangle breaking, the complexity */
/*   of the examination, and memory usage */

void moremagic(); /* Placeholder function */

#endif
```

```
/* FILENAME:  dr-namelist.h */
/**
 * dr-namelist
 * Dynamically Resizable Container For The Purpose Of
 * Collecting Globals In A Statement
 * Produced at Missouri S & T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifndef NAMELIST_H
#define NAMELIST_H

typedef struct namelist
{
  /* SCJ: A global that is a  list of pointers to
   pointers that store all the names */
  char** list;
  /*SCJ: All the name unique identifiers, to
   allow/disallow duplicates */
  int* nids;
  /*SCJ: The number of items currently stored in
   the name list */
  short size;
  /*SCJ: Then number of items the list could hold */
  short maxsize;
} namelist;

/**
 * Resizes A Name List
 *
 * Using vector style memory expansion
 *  resizes the name list
 *
 * param n The namelist to resize
 * return none
 */
void namelist_resize(struct namelist* n);
/**
 * Adds another item to the end of the namelist
 *
 * param n The namelist to add on to
 * param input The name of the item being added
 * param nid The id of the item being added.
 * return none
 */
void namelist_push(struct namelist* n, char* input, int nid);
```

```
/**
 * Pulls out a name based on an index
 *
 * param n The namelist to extract from
 * param index The item to extract
 * return NULL if it's out of bounds, otherwise, the index
 */
char* namelist_get(struct namelist* n, short index);
/**
 * Resets the namelist. Doesn't change its max size
 *
 * param n the namelist to reset
 * return none
 */
void namelist_clear(struct namelist* n);
/**
 * Makes a namelist if one doesn't already exist
 *
 * param n Checks if this is a namelist,
 *  if not, a new one is made
 * return A pointer to a new namelist, unless n is a namelist.
 */
struct namelist * namelist_makeonce(struct namelist* n);
/**
 * Computes a hash to describe the contents of an input
 *
 * param input The c-string to hash.
 * return an int that is a hash related to the input.
 */
int namelist_hash(char* input);
/**
 * Makes a new namelist and initializes it from an input string
 *
 * param newnamelist A pointer to a place where the pointer
 *  to the namelist can be stored
 * param in An input string to initialize the namelist with
 */
void namelist_new(struct namelist** newnamelist,char * in);
/**
 * Outputs a string made from the input namelist
 *
 * param n An input namelist to convert to string
 * return A c-string that is the raw contents of the namelist
 */
char* namelist_getraw(struct namelist* n);

#endif
```

```
/* FILENAME:  dr-types.h */

#ifndef DRUIDD_TYPES
#define DRUIDD_TYPES

/**
 * dr-type
 * Structures Used By DRUiDD
 * Produced at Missouri S&T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */
#include "dr-constants.h"

#define BREAKUPS 100

/**
 * A structure to store what nodes have been visited in a graph
 */
typedef struct VisitedList
{
  void * addr; /* A pointer to a node */
  struct VisitedList * nxt; /* The next item in this list */
} dbglist;

typedef struct VisitedList visitlist;

/* Persistant Structures - Structures used to store
information about string names etc */

/**
 * A list of the global variables used in a program
 */
typedef struct variableList {
  /* The name of the current Item in the list */
  char * name;
  /* The vid of the current item */
  int vid;
  /* The next item in the list */
  struct variableList * nxt;
  /* Enabled if it appears in a specified process */
  int appears[PROCS];
} variableList;

typedef struct intlist {
  int v; /* The value of this node */
  struct intlist * nxt; /* The next item in this list */
} ilist;
```

```
/* Data structures' definitions for geometric preprocessing. */

/* General Structures - Structures for storing critical
information in the finest container */
/**
 * Stores each branch of a branching structure
 */
typedef struct Segment
{
  int droot; /* David's root position */
  int root;  /* control pt. right before 'if' or 'do' */
  int branch;  /* control pt. which begins the option */
  int join;  /* control pt. right before 'od' or 'fi' */
  int isDo;  /* obvious  */
  int first; /* Is this the first control Structure */
  int last; /* Is this the last control structure */
  int depth; /* How nested is this structure */
  /* Stores how root should be broken up against a process */
  struct intlist *notches[PROCS];
  /* If this overflows, I will be impressed. */
  int breakup[PROCS][BREAKUPS];
  /* If this branch is nested, what segment is its parent? */
  struct Segment *parent;
  struct Segment *nxt;  /* for linked list */
} segment;

/**
 * A forbidden region along one axis
 */
typedef struct CriticalSection
{
  int p;  /* initial point of the section */
  int v;  /* terminal point of the section */
  int s;  /* identifier of the semaphore */
  struct CriticalSection *nxt;  /* for linked list */
}  critical;

/**
 * A general X Y point structure
 */
typedef struct  Point
{
  int x;  /* control pt. for first process */
  int y;  /* control pt. for second process */
  int isForbidden;  /* obvious */
  struct Point * nxt;
} point;
```

```
/**
 * Multiple Purpose: Forbidden Region, Component
 */
typedef struct  Rectangle
{
  point a;  /* minimal corner */
  point b; /* maximal corner */
  int isForbidden;  /* obvious */
  struct Rectangle *nxt;  /* for linked list */
} rectangle;

/**
 * Cross Section. Typically referred to as
 * the corner of a shell
 */
typedef struct  Xsection{
  segment * A;  /* interval in first process */
  segment * B;  /* interval in second process */
  /* Pointer to point that generated this cross section */
  point * pt;
  short using_vx; /* If this is created using a virtual */
  short using_vy; /* If this is created using a virtual */
  short jx; /* If this jumps forward or backward */
  short jy; /* If this jumps forward or backward */
  struct Xsection_link * nxt; /* Forward transition */
  struct Xsection_link * prev; /* Backward transition */
  int distance;
} xsection;

/**
 * A Temporary Structure To Simplify Implementation.
 * Usually Converted to Xsection
 */
typedef struct Xsection_corner
{
  /* Pointer to point that generated this cross section */
  struct Point * pt;
  struct Segment * A; /* interval in first process */
  struct Segment * B; /* interval in second process */
  short vx;  /* If this is created using a virtual */
  short vy; /* If this is created using a virtual */
  short jx; /* If this jumps forward or backward */
  short jy; /* If this jumps forward or backward */
  int distance; /* The enumeration of this structure */
  int count; /* An internal counter */
  struct Xsection_corner * nxt;
} corner;
```

```
/**
 * A structure that is also known as a shell
 */
typedef struct Xsection_link
{
  /* A link for parallel shells */
  struct Xsection_link * nxt;
  /* The inside corner */
  struct Xsection * x;
  /* The decompostion of the shells */
  struct Rectangle * decomp;
  int special; /* Flags for the shell */
} xsection_link;

/**
 * Structure for component graph entries
 */
typedef struct  NewTrans{
  /* Analogue of 'Trans', t_id will be the
   identifying index from the progGraph list.  */
  /* Note: this is a transition between components, */
  /* not between states (points). */
  unsigned int t_id; /* The id of the transition */
  struct Rectangle * A; /* The source rectangle */
  struct Rectangle * B; /* The destination rectangle */
  struct NewTrans * nxt; /* For hash collisions */
  int proc_i; /* The x direction process */
  int proc_j; /* The y direction process */
} newTrans;

typedef struct NewTrans dTrans;

/**
 * A structure for storing current N-dimensional figure
 */
typedef struct Cuboid
{
  /* The faces of the figure */
  struct NewTrans *face[PROC_PAIRS];
} Cuboid;

/**
 * A important point in the program execution
 */
typedef struct criticalChange
{
  int vid;
  /*The transition that this change takes place in */
  int segment;
  short op;
```

```
  /* 1 For Semas 0 for Globals */
  short type;
  /* If the segment jumps */
  short jumps;
  /* The next change of this type */
  struct criticalChange * nxt;
  /* The counter for this from the X axis */
  int distance;
  /* The segment that this is in */
  struct Segment * container;
} criticalChange;

/* Work Structures - Structures for storing partially
completed steps for recursive funcs, etc */

/**
 * Structure for helping count up virtuals
 */
typedef struct partitionRoot
{
  int vx;
  int vxs;
  int vmx;
} partitionRoot;




#endif
```

```
/* FILENAME:  dr-data.h */

#ifndef DATA_H
#define DATA_H

#include "dr-constants.h"
#include "dr-data.h"

/* Global data structures' instantiation. */

/* counter used to enumerate segments */
int sel;
/* contains info about segments in each process */
segment selection[PROCS][SEGS];


rectangle * rect[PROCS][PROCS] = {{NULL}};
xsection * partitionA[PROCS][PROCS] = {{NULL}};
point * pt[PROCS][PROCS] = {{NULL}};
dTrans * adjacency[ADJACENCY_SIZE] = {NULL};
rectangle * comp[PROCS][PROCS] = {{NULL}};


/*SCJ: */
/* List that gives the semaphores unique ids by their name */
variableList * gSemaphores = NULL;
/* List that gives the globals unique ids by their name */
variableList * gGlobals = NULL;
criticalChange * lockList[PROCS] = {NULL};
criticalChange * varList[PROCS] = {NULL};
short lockUsage[PROCS][SEMS] = {{0}};
int sels[PROCS] = {0};

int decount = 2;

#endif
```

```
/* FILENAME:  dr-visualize.h */

/**
 * dr-visualize
 * Visualization Functions Used In DRUiDD
 * Produced at Missouri S&T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */

#ifndef VISUALIZE_H
#define VISUALIZE_H

#ifndef VISUALIZE_STANDALONE
/* Mar 18: This adds Trailblazer support so we don't go
back and look for some DR resources */
#include "dr-main.h"
#else
extern int PROCS;
extern int SEMS;
extern int PROC_PAIRS;
#endif

#define VISUALIZE_STD_LOW 1
#define VISUALIZE_STD_HI 1
#define VISUALIZE_FBD_LOW 2
#define VISUALIZE_FBD_HI 7

#define VISUALIZE_SLEN 64

static char * visual_header[] =
{ /* Header for the GNU Plot File */
  "set terminal png transparent nocrop ",
  "enhanced font arial 8 size 800,600",
  "set title \"Two-Dim. Progress Graph Decomposition\"",
  "set style line 1 lc rgbcolor \"black\"",
  "set style line 2 lc rgbcolor \"forest-green\"",
  "set style line 3 lc rgbcolor \"blue\"",
  "set style line 4 lc rgbcolor \"orange\"",
  "set style line 5 lc rgbcolor \"purple\" pt 2 ps 1 lw 4",
  "set style line 6 lc rgbcolor \"magenta\"",
  "set style line 7 lc rgbcolor \"dark-pink\"",
  "set style line 8 lc rgbcolor \"green\" pt 2 ps 2 lw 4",
  "set style line 9 lc rgbcolor \"red\"",
  "set style line 10 lc rgbcolor \"dark-orange\"",
  "set style line 11 lc rgbcolor \"purple\" lw 4",
  /* The following are search order colorings */
  "set style line 12 lc rgbcolor \"red\"",
  "set style line 13 lc rgbcolor \"orange\"",
  "set style line 14 lc rgbcolor \"yellow\"",
```

```
  "set style line 15 lc rgbcolor \"green\"",
  "set style line 16 lc rgbcolor \"blue\"",
  "set style line 17 lc rgbcolor \"purple\"",
  "set style arrow 1 nohead ls 1",
  "set style arrow 2 nohead ls 2",
  "set style arrow 3 nohead ls 3",
  "set style arrow 4 nohead ls 4",
  "set style arrow 5 nohead ls 5",
  "set style arrow 6 nohead ls 6",
  "set style arrow 7 nohead ls 7",
  "set style arrow 8 nohead ls 10",
  "set style arrow 9 nohead ls 9",
  "set style arrow 10 nohead ls 11",
  "set style arrow 11 head ls 12 front filled",
  "set style arrow 12 head ls 13 front filled",
  "set style arrow 13 head ls 14 front filled",
  "set style arrow 14 head ls 15 front filled",
  "set style arrow 15 head ls 16 front filled",
  "set style arrow 16 head ls 17 front filled",
  "set style arrow 17 head ls 12 front",
  "set style arrow 18 head ls 13 front",
  "set style arrow 19 head ls 14 front",
  "set style arrow 20 head ls 15 front",
  "set style arrow 21 head ls 16 front",
  "set style arrow 22 head ls 17 front",
  "set style arrow 23 nohead ls 12 front",
  "set style arrow 24 nohead ls 13 front",
  "set style arrow 25 nohead ls 14 front",
  "set style arrow 26 nohead ls 15 front",
  "set style arrow 27 nohead ls 16 front",
  "set style arrow 28 nohead ls 17 front",
  "set grid",
  ""
};

#define STYLE_BRANCHES 3
#define STYLE_XLINE 9
#define STYLE_POINTS 8
#define STYLE_GAURD 4
#define STYLE_DROOT 9
#define STYLE_XSECTION 5
#define STYLE_SEARCH 11

#define STYLE_JUMP 23
#define STYLE_SEARCH_OPT 6
#define STYLE_SIMPLE 23

FILE * vis_gnup = NULL;
FILE * vis_points = NULL;
FILE * vis_corners = NULL;
```

```
FILE * vis_moves[PROC_PAIRS];
FILE * vis_deadlocks[PROC_PAIRS];
char vis_file_gnup[VISUALIZE_SLEN];
char vis_file_points[VISUALIZE_SLEN];
char vis_file_corners[VISUALIZE_SLEN];
char vis_file_out[VISUALIZE_SLEN];
char vis_file_moves[PROC_PAIRS][VISUALIZE_SLEN];
char vis_file_deadlocks[PROC_PAIRS][VISUALIZE_SLEN];
int current_i;
int current_j;
int visualize_arrow = 0;

#ifdef VISUALIZE_STANDALONE
void visualize_standalone_init()
{
  vis_moves = malloc(sizeof(FILE *)*PROC_PAIRS);
  vis_deadlocks = malloc(sizeof(FILE *)*PROC_PAIRS);
  vis_file_moves = malloc(sizeof(char *)*PROC_PAIRS);
  vis_file_deadlocks = malloc(sizeof(char *)*PROC_PAIRS);
  for(
}
#endif

#ifndef VISUALIZE_STANDALONE
/**
 * Initializes the variables and file accesses
 * used for the search visualization
 *
 * Opens and prepares a set of files for gnuplot to parse
 * and create a visualization of how the search proceeds
 * through the search space
 *
 * return none
 */
void visualize_search_init();
/**
 * Writes the location of a deadlock to the
 * deadlock datapoints file.
 *
 * param ttdepth the current position of each process
 *  at time of deadlock.
 *
 * return none
 */
void visualize_deadlock_point(int * ttdepth);
/**
 * Writes a move to the output file.
 *
 * May write a move to the output file based on the search
 * position the last time the function was called. If visited
```

```
 * is enabled, the arrow will be drawn differently to note
 * that the search hits a visited state. If draw is disabled
 * the position is only updated (For backwards moves)
 *
 * param ttdepth the current ttdepth (the previous ttdepth
 *  is stored as a static)
 * param visited whether the search is reporting that the
 *  end state has been visited
 * param draw whether the move should be drawn to the file.
 * return none
 */
void visualize_search_move(int * ttdepth,
short visited, short draw);
/**
 * Finishes and closes the visualization files.
 *
 * Writes the final statement needed to generate
 * the plot to output files, and closes all open files
 * used to perform visualization
 *
 * return none
 */
void visualize_search_finish();
#endif
/**
 * Initializes the process pair for partitioning visualization
 *
 * Partition visualization is done one pair at a time, so
 * each pair runs this function to prepare their set of files.
 *
 * param i the x direction process (DAC Enumeration)
 * param j the y direction process (DAC Enumeration)
 */
void visualize_init(int i, int j);
/**
 * Given an input list of rectangles, this draws all to output
 *
 * Draws rectangles, with cross lines to detect overlapping
 * rectangles (there should be none) forbidden rectangles are
 * colored based on how nested they are
 *
 * param tmpr a singly-linked list of rectangles to parse.
 */
void visualize_rectangle(struct Rectangle *tmpr);
/**
 * Records a critical point to the visualization
 *
 * param tmpp the point to record
 */
void visualize_point(point * tmpp);
```

```
/**
 * Records corner for a shell to the visualization output
 *
 * param tmpp a point to record as a corner
 */
void visualize_corner(point * tmpp);
/**
 * Given an array of branches, plots lines to represent
 * branch, join, root etc.
 *
 * Segment should be an array of branches,
 *  sel should be size of the list
 *
 * param branches An array of branch structures.
 * param sel the number of structures to consider
 * param x Whether the process is x or y axis
 */
void visualize_branching(segment * branches,int sel,int x);
/**
 * Finishes up partition visualization for a process pair.
 *
 * param i the x direction process
 * param j the y direction process
 */
void visualize_finish(int i, int j);
/**
 * Generates some bash scripts to make generating all
 * the plots a little quicker
 *
 * return none
 */
void visualize_generate_scripts();

#include "dr-visualize.c"

#endif
```

```
/* FILENAME:  dr-general.h */

/**
 * dr-general
 * General Transformation Functions For DRUiDD
 * Produced at Missouri S&T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */
#ifndef GENERAL_H
#define GENERAL_H

#include "dr-main.h"
#include <stdio.h>
#include <stdarg.h>

#define VERBOSITY_ALWAYS 0
#define VERBOSITY_WARN  1
#define VERBOSITY_COMP  1
#define VERBOSITY_ARCH  1
#define VERBOSITY_FCALL 2
#define VERBOSITY_LOOP  3
#define VERBOSITY_SUPER 3
#define VERBOSITY_HYPER 4 /* Replacement fo hyper verbosity */
#define VERBOSITY_DEBUG 5

extern short verbosity;

#ifdef VISUALIZE
/*PLACEHOLDER*/
void visualize_rectangle(struct Rectangle *tmpr);
#endif
/**
 * General Function
 *
 * Finds the smaller of the two numbers
 *
 * param a An integer
 * param b An integer
 * return the smaller of a and b
 */
int min(int a,  int b);
/**
 * General Function
 *
 * Find the factorial of a given input
 *
 * param a An integer
 * return a factorial
 */
```

```
int factorial(int a);
/**
 * General Function
 *
 * Finds the choose of a given input a choose b
 *
 * param a An integer
 * param b An integer
 * return Computes A choose B.
 */
int choose(int a,int b);
/**
 * General Function
 *
 * Recursively clears a singly linked list
 *
 * param start The head of the list to clean up.
 * return none
 */
void clear_slist(criticalChange * start);
/**
 * General Function
 *
 * Maintains a list of variable names and assigns
 * each var a unique ID
 *
 * pre a look up list initialized (Should be in init
 *  and a variable name to look up)
 * post adds the variable to the list if it
 *  wasn't already an item
 * param variable The name of the variable
 * param lookup The source to look up the variable from
 * param proc The process to consider
 * param critical The importance of the segment.
 * return the ID of the item or the added item
 */
int get_variableID(char * variable, variableList * lookUp,
int proc, short critical);
/**
 * General Function
 *
 * Looks up a variable via its variable ID and tells if
 * it is used in a specified process. No other changes made.
 *
 * param vid The variable ID to examine
 * param lookup The list to look up the variable in
 * param proc The process to consider
 * return True if the variable is used in the process
 */
int get_usageByVid(int vid, variableList * lookUp, int proc);
```

```
/**
 * General Function
 *
 * Finds and removes all the variables with ID -1 in
 * a given list of variables
 *
 * param head The Front of the list
 * return A list without any virtuals in it.
 */
criticalChange * clean_virtuals(criticalChange * head);
/**
 * General Function
 *
 * Sorts a list of critical changes by the
 *  segment no !!! they appear in
 * Places virtuals after real critical changes
 *
 * param head The front of the list
 * return The input list, but sorted.
 */
ilist * bs_intlist(ilist * head);
/**
 * General Function
 *
 * Sorts a list of critical changes by the
 *  segment no !!! they appear in
 * Places virtuals after real critical changes
 *
 * param head the front of the list
 * return A sorted list.
 */
criticalChange * bubble_sort(criticalChange * head);
/**
 * General Function
 *
 * Makes correction to handle multiple read writes at one
 * Segment, reducing it to most interesting type of change
 *
 * param a A list of changes to consider
 * param proc The process being parsed
 * return A pointer to the head of a reduced list.
 */
criticalChange * varlist_fix(criticalChange * a, int proc);
/**
 * General Function
 *
 * Takes two lists of critical changes and combines them.
 * Makes copies of every item in the list
 *
 * param a A list to merge
```

```
 * param b The other list to merge
 * return A list that is a merged version of A and B
 */
criticalChange * merge_lists(criticalChange * a,
criticalChange * b);
/**
 * General Function
 *
 * Remove items from a critical change list that do not appear
 * in the opposing process
 *
 * param x The list to parse
 * param j A process to consider
 * return A reduced list.
 */
criticalChange * remove_unused(criticalChange * x,int j);
/**
 * General Function
 *
 * Determines if specified critical change is used
 * in given process
 *
 * param c An interesting point to consider
 * param j The process to consider for
 * return True if the variable is not used in process j.
 */
int is_unused(criticalChange * c,int j);
/**
 * General Function
 *
 * Looks up a segment (branch of a control structure) based
 * on the transition segment and a given process
 *
 * param s The transisiton to look up the branch for
 * param proc The process of that transtion
 * return A pointer to the segment that contains that structure.
 */
segment * lookup_segment(int s,int proc);
/**
 * General Function
 *
 * Converts a corner to an xsection
 *
 * param r A list of corners to convert to Xsections
 * return none
 */
xsection * create_newXsection(corner * r);
/**
 * General Function
 *
```

```
 * Calls all other processing functions and guides through
 * the preprocessing process
 *
 * param nexus The starting point to work from.
 * param visited Input as NULL. Generated By Function.
 * return none
 */
int debug_count_trans(xsection * nexus, dbglist * visited);
/**
 * General Function
 *
 * Takes a target rectangle and a breaker. The intersecting
 * regions of the two rectangles are divided up. If forbid
 * is two then it removes a region.
 *
 * param target A list to break up by the breaker
 * param breaker A rectangle to slice up those in the target
 * param forbid Whether the breaking rectangle is forbidden
 *  or not. 2 Removes the region of breaker
 * return A pointer to the new head of the target list.
 */
rectangle * break_rect_byrect(rectangle * target,
rectangle * breaker, int forbid);
/**
 * General Function
 *
 * Returns a pointer to a rectangle made by 2 points.
 * Returns null if the rectangle would have a negative area
 *
 * param a The minimal corner.
 * param b The maximal corner.
 * return A pointer to a new rectangle
 */
rectangle * make_rectangle(point a, point b);
/**
 * General Function
 *
 * Same as make_rectangle, but takes in the
 * individual x and y values
 *
 * param x1 Minimal X coordinate
 * param y1 Minimal Y coordinate
 * param x2 Maximal X coordinate
 * param y2 Maximal Y coordinate
 * return A pointer to a new rectangle
 */
rectangle * make_rectangle_bycoord(int x1, int y1,
int x2, int y2);
/**
 * General Function
```

```
 *
 * Checks to see if a given rectangle has a negative area
 *
 * param in The rectangle to check against
 * param ident A name for the rectangle.
 * return none
 */
void debug_rect_check(rectangle * in, char * ident);
/**
 * General Function
 *
 * Counts the number of rectangles generated in decomposition
 * by secondary partioning, not the rectangles generated
 * previously for forbidden regions
 *
 * param nexus The starting point
 * param visited Give as NULL, used by recursion
 * return The number of xsections (including that one)
 */
int debug_count_rect(xsection * nexus, dbglist * visited);
/**
 * General Function
 *
 * Counts and prints the rectangles generated in
 * decomposition by secondary partitioning not the
 * rectangles generated previously for forbidden regions
 *
 * param nexus The starting X section
 * param visited An internal used in recursion
 * to prevent revists
 * return The number visited.
 */
int debug_print_rect(xsection * nexus, dbglist * visited);
/**
 * General Function
 *
 * Generates a has to describe a transition between two
 * rectangles based on the source rectangle
 *
 * param A The rectangle to hash
 * param proc_i The X direction process
 * param proc_j The Y direction process.
 * return The hash described by the rectangle
 */
unsigned int adj_hashRect(rectangle * A,
int proc_i, int proc_j);
/**
 * Handles printing messages approp. to desired verbosity
 *
 * There are 6 Levels of Verbosity
```

```
 * 0 - Squelch All Messages: Absolutely Nothing Prints
 *  To The Screen (For DR)
 * 1 - Show Warnings Only: Only show things like adjacency
 *  list failures, and occassional updates of progress
 *  through the search space for long searches
 * 2 - Show Some Preprocessing Information: Show interesting
 *  things like the number of components generated, How
 *  the branching structures are laid out
 * 3 - Show Some of The More Interesting Search
 *  Information: Show when big moves are made,
 *  when we change components, when tiny move fails etc.
 * 4 - Give Me Lists!: Starts Printing Out Some of the
 *  information that is stored as arrays/lists, including
 *  cuboid faces and other such things
 * 5 - Show me everything -- every Debug print
 *  statement still in the code!
 * Frequency of output flushes inversely prop. to error level:
 * This can be overriden with an always flush compiler option,
 *  or the never flush compiler option.
 * 0: Never Flush
 * 1: Flush Every 4096 (2^12)
 * 2: Flush Every  512 (2^9)
 * 3: Flush Every  64 (2^6)
 * 4: Flush Every  8 (2^3)
 * 5: Always Flush. (2^0)
 *
 * param level The message level
 * param format the message format to show
 * param ... the variables to print, just like printf
 * return none
 */
void dprintf(short level, const char * format, ...);

#include "dr-general.c"
#endif
```

```
/* FILENAME:  dr-analyze.h */

/**
 * dr-analyze
 * First stage processing for DRUiDD Project
 * Produced at the Missouri S&T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */
#ifndef ANALYZE_H
#define ANALYZE_H
#include "dr-main.h"
#include "dr-partitionsec.h"
/**
 * Processing Function (Pre-Partitioning)
 *
 * Calls all other processing functions and guides through
 * the preprocessing process
 *
 * return none
 */
void preprocess(void);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Initializes the variable list globals
 *
 * return none
 */
void init(void);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Early stage of preprocessesing that determines the tree
 * structure for a set of branches Using branches()
 *
 * param i the x-direction process
 * param j the y-direction process
 * return none
 */
void pairing(int i, int j);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Follows transition table of process, locating
 * branching structures, and generating a list of
 * critical points
 *
 * param i the x-direction process
 * param start Internal recursion counter,
```

```
 *  always start as zero
 * param stop Internal recursion counter,
 *  always start as zero
 * param sort Internal recursion counter,
 *  always start as one
 * return none
 */
void branches(int i, int start, int stop, short sort);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Generates interesting points for read writes of
 * globals and areas fobidden by Semaphore actions
 *
 * param i the x-direction process
 * param j the y-direction process
 * return none
 */
void corners(int i, int j);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Takes matches of locks to unlocks and generates
 * rectangles of forbidden regions which are stored
 * away for later
 *
 * return none
 */
void rectangles(int i, int j);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Takes an unlock and finds the closest unlock
 * Generates stdio warning if no match is found
 * (Which are frequent if the Dijkstra funcs
 * are processed)
 *
 * param i the x-direction process
 * param crit a list of critical changes to match locks
 *  and unlocks from
 * return none
 */
void matches(int i, critical** crit);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Finds all lock events in a given process
 *
 * param i the x-direction process
 * param lock An array to store the found locks to.
```

```
 * return none
 */
void locks(int i, criticalChange * lock[]);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Finds all unlock events in a given proc
 *
 * param i the x-direction process
 * param unlock An array to store the discovered unlocks to.
 * return none
 */
void unlocks(int i, criticalChange *  unlock[]);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Looks for critical intersections between write and read
 * actions between the two processes; generates and
 * stores a list
 *
 * param i the x-direction process
 * param j the y-direction process
 * return none
 */
void points(int i, int j);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Finds all global read operations in a given process
 *
 * param i the x-direction process
 * return A list of points where reads have been
 *  performed on globals
 */
criticalChange * reads(int i);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Finds all global write operations in a given process
 *
 * param i the x-direction process
 * return A list of points that writes on globals
 *  are performed
 */
criticalChange * writes(int i);
/**
 * Processing Function (Pre-Partitioning)
 *
 * Calls initial and secondary partitioning functions
 * to make the DRUiDD
```

```
 *
 * param i the x-direction process
 * param j the y-direction process
 * return none
 */
void partition(int i, int j);

#include "dr-analyze.c"
#endif
```

```
/* FILENAME:  dr-partitioninit.h */

/**
 * dr-partitioninit
 * Initial Partioning Functions For DRUiDD
 * Produced at Missouri S&T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */
#ifndef INITIAL_H
#define INITIAL_H

#include "dr-main.h"
/**
 * Preprocessing Function (Partitioning)
 *
 * Use minimal corners and positions of global conflict
 * points and branches to obtain a first approximation
 * of each cross-section. Equalize locks in 'parallel'
 * segments. Find x and y coordinates of minimal
 * corners of forbidden rectangles and global conflict
 * points. Use intervals of branches to equalize the
 * number of interactions. Do this recursively, and
 * then proceed with the partitioning from the origin.
 *
 * param i The X direction process
 * param j The Y direction process
 */
void initial_partition(int i, int j);
/**
 * Initial Partitioning Helper (Critical Change)
 *
 * Computes an equalized distance, that is, one
 * independent of the branching structure of the
 * program that each input is from the origin of
 * the program. Updates the input rather than
 * returning a changed set
 *
 * param x The input set of critical changes
 * param i The process being processed
 * param count The last depth before recursion
 * param current the current branching structure
 * return none
 */
void distance(criticalChange * x,int i,
int count, segment * current);
/**
 * Initial Partitioning Helper (Virtuals)
 *
 * Updates the stored branching structure to mark if
```

```
 * the section is first or last. In addition this adds
 * virtual points to make the partitioning structure
 * more neatly wrap around the branching structure of
 * the program. Virtuals are placed at the join of
 * each branching structure, as well as root and union
 *
 * param i The X direction process
 * param j The Y direction process
 * param x A list of currently existing marks along
 *  the i process
 * return A pointer to a head of an updated list.
 */
criticalChange * initial_partition_prerough(int i,
int j, criticalChange * x);
/**
 * Initial Partitioning Helper (Virtuals)
 *
 * initial_partition_rough takes an input root and
 * determines how many variables are used in each branch,
 * with each branch recursively partitioned for nesting
 *
 * param i The X direction process
 * param x A list of critical parition points along
 *  that process to consider
 * return A pointer to the head of an updated list.
 */
criticalChange * initial_partition_rough(int i,
criticalChange * x);
/**
 * Initial Partitioning Helper (Pre-Crosssection)
 *
 * Continues count operations along with
 * initial_partition_rough for the purpose of
 * generating virtuals
 *
 * param i The process to consider
 * param root The transition number of the root
 *  being considered
 * param x A list of critical partition points to consider
 * return the Number of changes within that root
 */
int initial_partition_root(int i, int root,
criticalChange * x);
/**
 * Initial Partitioning Helper (Pre-CrossSection)
 *
 * Does a rough job pretend pairing critical points
 * for the purpose of equalizing the critical
 * points across branching structures
 *
```

```
 * param i The process to consider.
 * param b The current input count
 * param c A list of critical change inputs
 * return A number of interactions within the region
 */
int rough_pairing(int i, int b, criticalChange * c);

/**
 * Initial Partition Helper (Pre-CrossSection [Corners])
 *
 * Generates a list of corners to be later converted to
 * cross-sections. These corners are seperate from those
 * produced by the corners function
 *
 * param x A list of interactions to make corners with
 *  in the X direction
 * param y A list of interactions to make corners with
 *  in the Y direction
 * param i The x direction process
 * param j The y direction process
 * return a list of corners to make in cross sections
 */
corner * initial_partition_xcorners(criticalChange * x,
criticalChange * y,int i, int j);
/**
 * Initial Partition Helper (Corner To Cross-Section)
 *
 * Converts a set of corners to an array of Cross-Sections
 * which are ready to be paired into shells
 *
 * param r an input list of corners to convert
 * param count the number of corners to conver
 * return A pointer to an array of cross sections
 */
xsection ** initial_partition_convert_corner(corner * r,
int count);
/**
 * Initial Partition Helper (Cross Section)
 *
 * Creates connections using the table of
 * unpaired cross sections.
 *
 * param parent The previous generated cross section
 * param r the cross sections that still need to be mapped
 * param visited cross sections that have been hit
 *  before (for branching)
 * param count the cross sections made so far
 * param distance the distance we are looking for the
 *  next cross-section to map
 * param i The X direction process
```

```
 * param j The Y direction process
 * return The number of sections created
 */
int initial_partition_partial_order(xsection * parent,
xsection ** r, short * visited, int count,
int distance,int i,int j);
/**
 * Initial Partition Helper (Cross-Section
 *
 * Follows the partitioning until a NULL is reached
 * and then attaches an outermost shell.
 *
 * param root The inside corner to attach to.
 * param tail The end shell to attach at the end
 * return none
 */
void initial_partition_attach_tail(xsection * root,
xsection * tail);
/**
 * Initial Partitioning Helper (Cross-Section)
 * (No Longer Used.)
 *
 * Moves the edges of cross sections that use
 * virtuals to their final location
 *
 * param The inside corner to consider snapping
 * return None
 */
void xsection_snap(xsection * nexus);
/**
 * [DEBUG] Initial Partitioning Helper (Cross-Section)
 *
 * Prints all cross sections from root without repeating
 *
 * param from The cross section before this one
 * param root The currect cross section
 * param count The number printed so far
 * param visited A container storing visited states so far
 * return None
 */
void print_xsections(xsection * from, xsection * root,
int count, xsection ** visited);

#include "dr-partitioninit.c"

#endif
```

```
/* FILENAME:  dr-partitionsec.h */

/**
 * dr-partitionsec
 * Secondary Partioning Functions For DRUiDD
 * Produced at Missouri S&T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */
#ifndef SECOND_H
#define SECOND_H

#include "dr-main.h"

/**
 * Secondary Partitioning
 *
 * Breaks cross sections into rectangles (components)
 * based on critical points
 *
 * param i The x direction process
 * param j The y direction process
 */
void secondary_partition(int i, int j);
/**
 * Secondary Partitioning Helper
 *
 * Loops through cross sections and adds components
 * for move to root
 *
 * param nexus The inside corner for this function call
 * param i The x direction process
 * param j The y direction process
 * param visited A structure of visited nexuses.
 * return none
 */
void secondary_branch_helper(xsection * nexus,
int i, int j,visitlist * visited);
/**
 * Secondary Partitioning Helper
 *
 * Takes a cross section and "adds blue lines,"
 * making the 3 basic components that form the
 * cross-section and stores them to that cross section
 *
 * param nexus The inside corner for the cross section
 * param i The X direction process
 * param j the Y direction process
 * param visited A container to keep track of visited Nexus
 */
```

```
void addBlueLines(xsection * nexus,
int i, int j, visitlist * visited);
/**
 * Secondary Partitioning Helper
 *
 * Removes overlapping regions because of the branching
 * structure of the program. Also removes gaps that don't
 * need components because search never passes through them.
 *
 * param nexus The inside corner for the cleanup
 * param i The X direction process
 * param j The Y direction process
 * return none
 */
void xsect_cleanup(xsection * nexus, int i, int j);
/**
 * Secondaray Partitioning Helper
 *
 * Breaks down regions based on the minimal and maximal
 * corners of the forbidden rectangles
 *
 * param nexus The inside corner of the cross sections
 * param i The X direction process
 * param j The Y direction process
 */
void secondary_forbidden_decompose(xsection * nexus,
int i, int j);
/**
 * Secondary Partitioning Helper
 *
 * Determines if a point is contained within a
 * specified set of decompositions
 *
 * param p The point to check
 * param r The set of rectangles a point could be within
 */
short point_in_decomposition(point p, rectangle * r);
/**
 * Secondary Partitioning Helper
 *
 * Breaks down regions based on critical points
 * within that region
 *
 * param nexus The inside corner of the region
 * param i The X direction process
 * param j The Y direction process
 * param visited A container for visited inside corners
 */
void secondary_point_decompose(xsection * nexus,
int i, int j, visitlist * visited);
```

```
/**
 * Secondary Partitioning Helper
 *
 * Loops through the cross sections and eliminates regions
 * which has been reduced to zero area to prevent uneeded
 * entries in the adjacency table
 *
 * param nexus The inside corner of the region to check
 * param visited A container holding the visited states
 *  that have been searched through
 * param i The X direction process
 * param j The Y direction process
 */
int zero_area_cleanup(xsection * nexus,
visitlist * visited, int i, int j);

#include "dr-partitionsec.c"

#endif
```

```
/* FILENAME:  dr-graph.h */

/**
 * dr-graph
 * Functions For Generating Directed Component Graph
 * For DRUiDD
 * Produced at Missouri S&T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */
#ifndef GRAPH_H
#define GRAPH_H

#include "dr-main.h"

/**
 * Processing Function (Post-Partitioning)
 *
 * Analyze lists of components to determine adjacency
 *
 * param i the x-direction process
 * param j the y-direction process
 * return none
 */
void newtable(int i, int j);
/**
 * Processing Function (Post-Partitioning)
 *
 * Measures the standard deviation of the adjacency table.
 * Standard deviation is a good measure of how well
 * distributed the table is.
 *
 * param list The list of numbers to determine the
 *  Standard Dev of
 * param count The number of items in the list.
 * return The standard deviation of the input list
 */
double adj_debug_std(int * list, int count);
/**
 * Processing Function (Post-Partitioning)
 *
 * Uses the adj_debug_std function to perform a
 * measurement of how well the has function performs
 *
 * return none
 */
void adj_debug_distribution();
/**
 * Processing Function (Post-Partitioning)
 *
```

```
 * Sets up selectign two lists of rectangles to pair
 *
 * param nexus The Start cross section
 * param visited Internal counter for visited states.
 * param i the x-direction process
 * param j the y-direction process
 * return The visited list
 */
visitlist * newtable_do(xsection_link * nexus,
visitlist * visited, int i,int j);
/**
 * Processing Function (Post-Partitioning)
 *
 * rearranges the xsection rectangle list to put it
 * into comp, a list of all compositionswithout any
 * adjacency information, used to allow lookup of
 * compositions when the adjacency list does not
 * work out.
 *
 * param in A list of rectangles to add to the the
 *  comp global
 * param i The X direction process
 * param j The Y direction process
 * return none
 */
void newtable_full_comp(rectangle * in, int i, int j);
/**
 * Processing Function (Post-Partitioning)
 *
 * Checks to see if two rectangle lists have any items
 * which are adjacent and adds their entries to the
 * adjacency table
 *
 * param tmpA The rectangle that would be the source
 * param tmpB The rectangle that would be the destination
 * param i the x-direction process
 * param j the y-direction process
 * return none
 */
void newtable_rectcombo(rectangle * tmpA,
rectangle * tmpB,int i, int j);
/**
 * Processing Function (Post-Partitioning)
 *
 * Determines if two rectangles share any edges
 *
 * param A The rectangle that would be the source
 * param B The rectangle that would be the destination.
 * param i the x-direction process
 * param j the y-direction process
```

```
 * return none
 */
short is_adjacent(rectangle * A, rectangle * B,int i,int j);
/**
 * Processing Function (Post-Partitioning)
 *
 * Places a dTrans into the adjacency table.
 *
 * param a A transition to place into the adjacency table.
 * return The index of the inserted item
 */
unsigned int adj_HashRectTrans(dTrans * a);

#include "dr-graph.c"

#endif
```

```
/* FILENAME:  dr-explore.h */

/**
 * dr-explore
 * State Space Search For the DRUiDD Project
 * Produced at Missouri S&T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */
#include "dr-main.h"
#ifndef DRUIDDEXPLORE_H
#define DRUIDDEXPLORE_H
extern int visualize_arrow;
unsigned int reccount = 0;
dTrans * dtrns_tmp;
unsigned short HIT_MAX_DEPTH = 0;
/**
 * new_state wrapper to make some setup things simpler
 *
 * Simplifies the first call to the new_state.
 *
 * return none
 */
short dac_new_state();
/**
 * Main new_state function for the DR partition method search
 *
 * This function performs several action. On specific calls
 * it  * will select a new cuboid without previous adjacency
 * information (For the purpose of starting the search as well
 * as resetting the wrap variable [Which detects deadlock]).
 * It performs two functions: big_move and tiny_move. See their
 * descriptions for additional information!
 *
 * param pcuboid the cuboid that the search position resided in
 *  before this call. If NULL, a new cuboid will be selected
 *  without any adjacency info.
 * param wrap Used to detect a state where no valid exit move
 *  exists for this set of processes, when all non semaphore
 *  processes are one, it is a DL state. NULL will reset
 *  the wrap state
 * return A code representing the successfulness of the
 *  new_state call.
 */
int d_newstate(const Cuboid * pcuboid, short * wrap);
/**
 * Big Move Function. Performs a large multiple transition move.
 *
 * Big move makes a large, multiple transition move to arrive
 * at the far edge of a component, but not forcing a new one
```

```
 *
 * param II the process to do the large move on
 * param distance the number of steps to take
 * param order order number for the purpose of visualization
 * return code showing the status of the big move,
 *  0 for failure, 1 for success
 */
int d_move(int II, int distance, int order);
/**
 * Performs a tiny move, which should force a new component
 * for some cuboid.
 *
 * Tiny move will make a step of a single direction of a
 * specified process which should force a new component
 * for the next level of depth. If the tiny step moves to
 * a root, this function also handles considering each of
 * the guards.
 *
 * param II the process to take the tiny step with
 * param cuboid the cuboid the search currently resides in
 * param wrap the current wrap status
 * param order the visualization colorization parameter.
 * return the status of the move. 0 for failure.
 */
int tinymove(int II, const Cuboid *cuboid,
short * wrap, int order);
/**
 * Takes a target position and removes items until the
 * target is reached.
 *
 * Removes items from the trail stack until the target
 * trail pointer is reached can rewind all the way to
 * the starting depth
 *
 * param ttrpt the target trail position to rewind to
 * return returns the restored trail pointer
 */
Trail * d_umove(Trail * ttrpt);

#include "dr-explore.c"

#endif
```

```
/* FILENAME:  dr-search.h */

/**
 * dr-search
 * Search Helping Functions For DRUiDD
 * Produced at the Missouri S&T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */
#ifndef SEARCH_H
#define SEARCH_H
#include "dr-main.h"

extern double adjacency_failures;

/**
 * Updates the specified pair face of the current cuboid.
 *
 * Given the current face of a cuboid, the process pair
 * it represents, this function will return a pointer to
 * a new face (That is adjacent to the input) that contains
 * the defined point. This function will return NULL if
 * there is no face that qualifies in the adjacency list
 *
 * param i the i direction process, DAC enumeration
 * param j the j direction process, DAC enumeration
 * param pi the postion in the i process
 * param pj the postion in the j process
 * param current the face that was the correct position
 *  before a move caused this evaluation
 * return An adjacency entry that defines the current
 *  position.
 */
dTrans * update_face(int i, int j, int pi, int pj,
dTrans * current);
/**
 * Finds a face given the process pair and the pistion
 * within each process.
 *
 * This function is similar to update_face() but instead
 * of using the adjacency list to find a face, this
 * searches the full list of qualifying components for
 * a face that fits, and is generally slower. Makes an
 * entry in the table to make successive lookups faster.
 *
 * param i the i direction process, DAC enumeration
 * param j the j direction process, DAC enumeration
 * param pi the postion in the i process
 * param pj the postion in the j process
 * param a the source rectangle
```

```
 * return a transition that defines the current position.
 **/
dTrans * update_face_force(int i, int j,
int pi, int pj, rectangle * a);
/**
 * Given an imput component, this function will find or
 * create an adjacency entry that starts with the
 * specified rectangle
 *
 * If this function is called without a and no entry is
 * found that starts with b it will make an entry in the
 * list where it is adjacent to itself. Ideally, one should
 * try and give it a second rectangle a, that was the source
 * of the move. That is, the search went from a to b (a->b)
 *
 * param b an input rectangle to locate
 * param a the source adjacency rectangle. May be NULL.
 * param i the i direction process, DAC enumeration
 * param j the j direction process, DAC enumeration
 * return an adjacency entry for the given rectangle,
 **/
dTrans * rectangle_find_adjacency(rectangle* b, int i, int j);
dTrans * rectangle_find_adjacency2(rectangle* b, rectangle* a,
int i, int j);
/**
 * Finds an adjacency entry that starts with rectangle
 *
 * param a the start rectangle of the entry
 * param i the i direction process
 * param j the j direction prcesss
 */
dTrans * rectangle_to_adjacency(rectangle * a, int i, int j);
/**
 * Gives an a current cuboid and a set of depths, gives back
 * a cuboid that describes the current position.
 *
 * param ttdepth The current depth of each process in program
 * param current The previous cuboid
 * return A cuboid that represents the position described
 *  through ttdepth
 */
Cuboid select_cuboid(int * ttdepth, const Cuboid * current);
/**
 * Discovers a cuboid face for an a process pair at a
 * specified postion
 *
 * param i The X direction process
 * param j The Y direction process
 * param idepth the current position of i direction process
 * param jdepth the current position of j direction process.
```

```
 * return a pointer to a face that fits
 */
rectangle * discover_face(int i, int j, int idepth, int jdepth);
/**
 * DFS Helper
 *
 * Checks the rectangle against the depth. If the rectangle is
 * not within the area the function returns false
 *
 * param i The X direction process
 * param j The Y direction process
 * param idepth The position along the X-axis
 * param jdepth The position along the Y-axis
 */
short check_rect_tt(rectangle * A, int i, int j,
int idepth,int jdepth);

#include "dr-search.c"

#endif
```

```
/* FILENAME:  dr-time.h */

/**
 * dr-time
 * Function Timing Functions
 * Produced at Missouri S&T June 2009 - Present
 * D. Cape <dacvdc@mst.edu>, S. Jackson <scj7t4@mst.edu>,
 * B. McMillin <ff@mst.edu>
 */
#ifndef DRUIDD_TIME
#define DRUIDD_TIME
#include "dr-main.h"
#include <sys/time.h>
#include <sys/types.h>
#include <time.h>

#define FUNCTIONS 51

/**
 * A structure to store a time counter
 */
typedef struct func_timer {
  /* The time the timer was last started */
  struct timeval start;
  /* The time the timer was last stopped */
  struct timeval finish;
  /* The total time the timer has counted so far */
  double total;
  /* If the timer is currently started */
  unsigned short open;
} func_timer;

static char * func_list[] = {
  "run", /* 0 */
  "new_state", /* 1 */
  "preprocess", /* 2 */
  "pairing", /* 3 */
  "branches", /* 4 */
  "corners", /* 5 */
  "rectangles", /* 6 */
  "matches", /* 7 */
  "locks", /* 8 */
  "unlocks", /* 9 */
  "points", /* 10 */
  "reads", /* 11 */
  "writes", /* 12 */
  "partition", /* 13 */
  "initial_partition", /* 14 */
  "distance", /* 15 */
  "initial_partition_rough", /* 16 */
```

```
  "initial_partition_root", /* 17 */
  "rough_pairing", /* 18 */
  "initial_partition_xcorners", /* 19 */
  "initial_partition_convert_corner", /* 20 */
  "initial_partition_partial_order", /* 21 */
  "initial_partition_attach_tail", /* 22 */
  "xsection_snap", /* 23 */
  "secondary_partition", /* 24 */
  "addBlueLines", /* 25 */
  "zero_area_cleanup", /* 26 */
  "secondary_point_decompose", /* 27 */
  "secondary_forbidden_decompose", /* 28 */
  "newtable", /* 29 */
  "newtable_full_comp", /* 30 */
  "newtable_rectcombo", /* 31 */
  "select_cuboid", /* 32 */
  "discover_face", /* 33 */
  "break_rect_byrect", /* 34 */
  "do_the_search", /* 35 */
  "xsect_cleanup", /* 36 */
  "main", /* 37 */
  "dac_new_state", /* 38 */
  "d_newstate", /* 39 */
  "d_move", /* 40 */
  "tinymove", /* 41 */
  "d_umove", /* 42 */
  "part of d_newstate", /* 43 */
  "another part of d_newstate", /* 44 */
  "another part of d_newstate", /* 45 */
  "another part of d_newstate", /* 46 */
  "another part of d_newstate", /* 47 */
  "another part of d_newstate", /* 48 */
  "another part of d_newstate", /* 49 */
  "rough_pairing+initial_partition_root", /* 50 */
  0
};

func_timer functime[FUNCTIONS];

/**
 * Prepares all timers
 *
 * Prepares N timers where N is the value of FUNCTIONS
 *
 * return none
 */
void init_timers();
/**
 * Starts a timer given by id
 *
```

```
 * Starts timer and marks it as open
 *
 * param id The timer to start
 * return none
 */
void start_timer(int id);
/**
 * Stops an open timer and adjusts the total amount
 * of time recorded by it
 *
 * param id The timer to stop
 * return none
 */
void stop_timer(int id);
/**
 * Closes all open timers
 *
 * return none
 */
void close_open_timers();

#include "dr-time.c"

#endif
```

BIBLIOGRAPHY

[1] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Prentice Hall, 2003.

[2] N. Rescher and A. Urquhart, *Temporal Logic*. Springer-Verlag, 1971.

[3] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[4] E. Clarke, Jr., O. Grumberg, and D. Peled, *Model Checking*. The MIT Press, 1999.

[5] G. Holzmann, *The Spin Model Checker Primer and Reference Manual*. Addison-Wesley, 2004.

[6] C. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[7] E. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453 – 457, 1975.

[8] K. McMillan, *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1992.

[9] E. Goubault and M. Raussen, "Dihomotopy as a Tool in State Space Analysis: Tutorial," in *Proceedings of the 5th Latin American Symposium on theoretical informatics, S. Rajsbaum, Ed.*, pp. 16–37, Springer-Verlag, April 03 - 06, 2002.

[10] S. Gradara, A. Santone, and M. Villani, "DELFIN$^+$: An Efficient Deadlock Detection Tool for CCS Processes," *Journal of Computer and System Sciences*, vol. 72, no. 8, pp. 1397–1412, 2006.

[11] A. Mazurkiewicz, "Trace theory," in *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*, (New York, NY, USA), pp. 279–324, Springer-Verlag New York, Inc., 1987.

[12] S. Carson and P. Reynolds, Jr., "The Geometry of Semaphore Programs," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 1, pp. 25–53, 1987.

[13] V. Pratt, "Higher-dimensional Automata Revisited," *Mathematical Structures in Computer Science*, vol. 10, no. 4, pp. 525–548, 2000.

[14] R. van Glabbeek, "On the Expressiveness of Higher Dimensional Automata," *Theor. Comput. Sci.*, vol. 356, no. 3, pp. 265–290, 2006.

[15] E. Goubault and E. Haucourt, "A Practical Application of Geometric Semantics to Static Analysis of Concurrent Programs.," in *CONCUR*, pp. 503–517, 2005.

[16] L. Fajstrup, M. Raussen, and E. Goubault, "Algebraic Topology and Concurrency," *Theor. Comput. Sci.*, vol. 357, no. 1, pp. 241–278, 2006.

[17] M. Raussen, "Deadlocks and Dihomotopy in Mutual Exclusion Models," *Theor. Comput. Sci.*, vol. 365, no. 3, pp. 247–257, 2006.

[18] E. Coffman, M. Elphick, and A. Shoshani, "System Deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, 1971.

[19] D. Cape, B. McMillin, B. Passer, and M. Thakur, "Inductive versus Recursive Partitioning of Progress Graphs," tech. rep., Department of Computer Science, University of Missouri–Rolla, Rolla, MO, July 2007.

[20] B. Bollig and M. Leucker, "Deciding LTL over Mazurkiewicz Traces," *Temporal Representation and Reasoning, International Symposium on*, vol. 0, pp. 189–197, 2001.

[21] J. Yeh, *Real Analysis: Theory of Measure and Integration, Second Edition.* World Scientific Publishing Co. Pte. Ltd., 2006.

[22] D. Cape, B. McMillin, B. Passer, and M. Thakur, "Recursive Decomposition of Progress Graphs," in *Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, pp. 23–31, 2009.

[23] D. Cape and B. McMillin, "Dihomotopic Reduction Used in Deadlock Detection," in *33rd Annual IEEE International Computer Software and Applications Conference*, pp. 648–651, 2009.

[24] D. Cape, S. Jackson, and B. McMillin, "Dihomotopic Deadlock Detection via Progress Shell Decomposition," in *Second International Conference on Advances in System Testing and Validation Lifecycle*, 2010, to appear.

[25] G. Holzmann, *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[26] D. Cape, S. Jackson, and B. McMillin, "Visualization of Dihomotopic Deadlock Detection via Progress Shell Decomposition," tech. rep., Undergraduate Research Symposium, Missouri University of Science and Technology, Rolla, MO, 2010, to appear.

VITA

David Andrew Cape was born in Pensacola, Florida, on May 18, 1966. His family soon relocated to the suburbs of Saint Louis, Missouri, however, where he spent his childhood. He was fortunate to attend John Burroughs School there for three years, then graduated from Holland Hall School in Tulsa, Oklahoma. He was a National Merit Scholar and earned his bachelor of arts degree in mathematics/physics at New College of Florida.

As a mathematics student at the Massachusetts Institute of Technology, he developed an interest in algebraic topology, algebraic geometry, and number theory. He took a linear programming course at Washington University in Saint Louis and then earned his master of science degree in mathematics at the University of Florida in 1994. As a doctoral student there, his study of non-Archimedean analysis was guided by Professor Richard M. Crew. He earned his master of science degree in computer science in 2006 at the University of Missouri-Rolla (recently renamed Missouri University of Science and Technology).

Before beginning his study of computer science, he was an adjunct instructor of mathematics at East Central College in Union, Missouri. Courses that he has taught as a graduate assistant include algebra, trigonometry, calculus, data structures, and algorithms. At the present time, software modeling and verification (especially in relation to the security of distributed and cyber-physical systems), computational complexity, and evolutionary algorithms are his primary professional interests.