ITR: Automated Verification of Asynchronous Software Systems

Gianfranco Ciardo* Gerald Luettgen†

Project Summary

The proposed research is devoted to the development and implementation of novel sequential and parallel algorithms for the verification of asynchronous software systems, such as communication protocols and distributed or embedded software. Existing automated techniques to verification via state–space exploration, in particular symbolic model checking based on Binary Decision Diagrams (BDDs), focus on verifying synchronous hardware and software. Although symbolic model checking may in principle be applied to debugging asynchronous software systems, this poses new challenges which are not or only insufficiently addressed in the literature. Most importantly, the inherent complexity of asynchronous software makes state–space exploration a time–bound problem, in addition to a memory–bound problem. The proposed work addresses these two fundamental limitations by developing algorithms that employ Multi–valued Decision Diagrams (MDDs) and Boolean Kronecker operators to encode sets of states and transitions, respectively, in contrast to BDDs traditionally used for both purposes. This paves the way for exploiting the property of event locality that is inherent in asynchronous software and, thereby, for greatly improving the efficiency of sequential algorithms and enabling their efficient parallelization. The project will focus on the following lines of inquiry:

1. The investigation of design choices for sequential MDD–based state–space generation algorithms exploiting event locality, and the development of sequential model checkers on top of these algorithms.

2. Shared–memory and distributed–memory parallelizations of the above algorithms and of the according model checkers.

3. The conduct of case studies involving access protocols in distributed B–tree algorithms and avionics systems to assess the utility of the work.

This research will greatly enhance the applicability of state–space exploration techniques, especially symbolic model checking, to the verification of asynchronous software systems. The implementation of the derived algorithms in form of C++ packages will allow an easy integration and comparison with other existing verification and validation tools, as well as with industrial design tools. The packages will be made web–accessible for remote execution at the College of William and Mary, in order to gather run–time statistics that will enable to fine–tune its performance with increasing experience. Last, but not least, this project is an excellent opportunity for exposing graduate and undergraduate students to advanced software engineering technologies, as it combines the important topics of parallel and distributed algorithms and state–of–the–art verification and debugging techniques, whose industrial relevance is on the rise.

*Department of Computer Science, The College of William and Mary, Williamsburg, VA 23187–8795. E–mail: ciardo@cs.wm.edu. URL: www.cs.wm.edu/~ciardo (Principal Investigator)

†Department of Computer Science, The University of Sheffield, Regent Court, 211 Portobello Street, Sheff fi eld S1 4DP, UK. E–mail: g.luettgen@dcs.shef.ac.uk. URL: www.dcs.shef.ac.uk/~luettgen (International Collaborator).
1 Overview

A key component of today’s information technology is the software for distributed embedded systems; examples include mobile applications such as communication and electronic commerce protocols [54, 86], as well as avionics and automotive systems [101, 119]. Despite their growing presence, distributed embedded systems are difficult and expensive to design and debug. The main challenge lies in the inherent asynchrony of distributed systems, which may result in subtle and often unanticipated interactions of system components. Thus, the quality assurance of these systems currently relies on extensive testing. Indeed, industrial practice shows that testing budgets represent about half of the development costs. Nevertheless, system errors still arise with sometimes devastating consequences, either financially or for personal safety [100].

Formal verification offers an appealing approach to complement testing with mathematical methods that determine whether a system satisfies its requirements. Formal techniques, relatively new to software engineering, are successfully used in other engineering disciplines. In fluid dynamics, for example, computational airframe models enable engineers to quickly analyze the quality of different designs. Although this approach does not make the testing of the real product obsolete, it reduces the need for expensive wind–tunnel testing. Analogously, formal verification can benefit the development of software by assisting in quickly revealing errors, omissions, and inconsistencies in designs and implementations. Research in this area has already led to the emergence of fully–automated verification techniques such as temporal–logic model checking [52, 56, 109, 136], which help make formal methods accessible to non-specialists. The growing number of case studies involving commercial products (see [58] for an overview) and the fact that several vendors, e.g., Cadence, Synopsys, and Telelogic, sell tools with formal verification support attest to this development.

Verification of asynchronous software systems. Despite the evident advantages of using automated formal verification in system development, these techniques are very sparsely employed in information technology applications, especially in the development of software for distributed embedded systems. To understand the reason for this, we consider the model–checking approach in more detail. Model checking involves the use of decision procedures to automatically determine whether a finite–state machine satisfies a specification given as temporal–logic formula. This is done by systematically generating and exploring the state space of the system under consideration. Since state spaces of real–world software systems tend to be very large, symbolic representations are frequently used for storing and manipulating them efficiently. Of particular importance for symbolic representations are Binary Decision Diagrams (BDDs) [1, 27, 28], which triggered the success of model checking in the hardware industry, as recent articles in the EE Times show [79, 140]. Indeed, the behavior of synchronous digital circuits can straightforwardly be encoded in BDDs.

However, BDD–based model checking [29, 34, 112] faces serious problems when applied to software systems, especially to asynchronous software. The reason for this is the inherent higher complexity of distributed systems, as their state spaces tend to explode with the number of system components. This does not only lead to BDDs that are often too large to fit in the memory of a single workstation, but it also requires a fair amount of time for constructing the state spaces. In fact, many engineers use model checking as an advanced debugging technique and expect feedback within minutes, not hours [89]. Hence, when considering asynchronous software, state–space generation and exploration become time–bound problems, in addition to memory–bound problems. Although several efforts have been made to parallelize algorithms for generating and manipulating BDDs on shared–memory and distributed–memory architectures, thereby utilizing the larger memory and higher computation power available on those machines, little or no speed–ups have been achieved. Recent work by the PI of this proposal and his international collaborator, however, shows potential for very significant improvements for the class of asynchronous software systems [38]. This is due to the following observations:
Multi–valued Decision Diagrams (MDDs) [95, 117] and Boolean Kronecker operators [4, 23, 31] can often be much more compact than BDDs in encoding the state space and next–state function of a globally–asynchronous locally–synchronous system, in particular distributed embedded software.

Asynchronous software — in contrast to hardware or synchronous software — has an inherent notion of locality. We refer to this locality as event locality [40], as asynchronous software is usually event–driven. Event locality enables a new class of fundamentally different approaches to symbolic manipulation that have so far not been explored in the literature.

Our results show that a prototypic sequential algorithm for state–space generation based on these ideas is several orders of magnitude faster and more memory–efficient than existing BDD–based algorithms [38, 46, 47, 117]. While these preliminary achievements are impressive, our findings have also introduced many novel research questions and challenges which deserve further investigation; this proposal elaborates on them.

Research agenda. The goal of this research project, which involves a collaboration between the PI at the College of William and Mary and an international collaborator at the University of Sheffield, UK, is to develop mathematically well–founded and practically useful algorithms for constructing and exploring state–spaces of asynchronous software systems. The specific topics to be investigated include the following.

MDD–based algorithms for state–space generation and model checking. We propose to develop and implement time– and space–efficient sequential state–space generation and model–checking algorithms for asynchronous software systems. The novelty of these algorithms, compared to approaches reported by other researchers, will be the use of MDDs and Kronecker operators instead of the much more common BDDs and, most importantly, the exploitation of event locality and iteration control to affect the order in which MDD nodes are explored.

Parallel implementations of the algorithms. We will investigate parallelizations of the above algorithms for shared–memory and distributed–memory architectures. In contrast to related work, the parallelization will be done along the notion of event locality. We expect that this will significantly speed–up state–space generation while utilizing the larger memory available on parallel machines, in particular PC clusters and networks of workstations. Major issues to address concern load balancing and cache consistency in the shared–memory case, and additionally message overhead reduction and termination detection in the distributed–memory case.

Tool integration and case studies. We will integrate the sequential and parallel algorithms — as well as the symbolic model checker built on top of them — in the tool SMART [39]. This will allow us to conduct two case studies to evaluate all aspects of our research. The first case study deals with the verification of new speculative parallel B–tree algorithms. The second case study is concerned with applying symbolic model checking to reveal deficiencies in the human–computer interaction between aircraft pilots and digital flight decks.

The proposed research will greatly enhance the applicability of state–exploration techniques and symbolic model checking to the verification and analysis of asynchronous software systems. The implementation of the derived algorithms as C++ packages will allow for an easy integration in other existing verification and validation tools, as well as with industrial design tools. Moreover, all algorithms will be made web–accessible for remote execution at the College of William and Mary, thereby promoting their uptake and dissemination in the wider research community. We plan to complement the project with courses, seminars, and reading classes for graduate students. The proposed research is an excellent basis for teaching, as it combines the two important topics of parallel and distributed algorithms and advanced software engineering technologies for debugging and verifying asynchronous software.
2 Background on State–space Generation and Exploration

Sequential techniques and algorithms for state–space generation and exploration can be classified according to whether they employ explicit or symbolic representations of state spaces. Of particular importance to this proposal are symbolic algorithms as well as the relatively few published approaches to their parallelization.

2.1 Sequential Explicit Techniques

Explicit techniques represent state spaces by trees, hash tables, or graphs, where each state corresponds to an entity of the underlying data structure [30, 40, 67, 87]. Thus, the memory needed to store the state space of a system is linear in the number of the system’s states, which in practice limits explicit state–space techniques to fairly small systems having at most a few million states.

For state–space generation, however, reduction techniques may be applied to reduce the sizes of state spaces. Such techniques minimize state spaces with respect to behavioral equivalences [69], interface specifications [81], or partial orders [55, 77, 78, 131, 132, 159] inherent in the interleaving semantics for asynchronous systems. Other techniques, such as bit–state hashing [88], focus on storing states using as little memory as possible.

For state–space exploration, the most popular technique is temporal–logic model checking [52, 56, 136]. Temporal logics [70, 71, 108, 154] support the formulation of assertions about a system’s behavior evolving over time. To verify a system, one specifies its desired properties as formulas that the system should satisfy. These include safety properties, defining what should always be true of a system, and liveness properties, describing conditions that a system must eventually satisfy. For example, one might require that a communications protocol is always deadlock–free (a safety property) and that it eventually delivers messages (a liveness property). Temporal logic has been the subject of intensive research in the past twenty years. Numerous variants, including Computation Tree Logic (CTL) [52], CTL* [71], the µ–calculus [98, 154], and Linear–time Temporal Logic (LTL) [160], have been designed and techniques for proving systems correct have been developed [84, 109]. When the systems considered are finite–state, “proofs of correctness” may be conducted automatically using decision procedures known as model–checking algorithms, implemented in model checkers. Several such approaches working on explicit representations of system models have been introduced [17, 51, 52, 61, 76, 78, 104, 160], and various case studies testify to their practical utility [58]. For good surveys on temporal logics and model checking, see [70, 154] and [56, 109], respectively.

To improve efficiency, model–checking tools [59, 60, 67, 87] often employ partial–order techniques [75, 77, 130], reduction methods based on exploiting symmetries of state spaces [53], or both [72]. Moreover, compositional [8, 57, 99, 148] and so–called partial [6] approaches to model checking have been developed, to avoid the need to construct the overall state space of a distributed system by gradually analyzing its components. Finally, it should be mentioned that explicit techniques for state–space generation have proved an especially useful first step if one is interested in the numerical analysis of Markov processes defined over such state spaces [31, 96].

2.2 Sequential Symbolic Techniques

Symbolic techniques allow one to store and manipulate sets of states in sublinear space. Most of them rely on Binary Decision Diagrams (BDDs) to represent Boolean functions efficiently [1, 27], into which state spaces can be mapped. The advent of BDD–based techniques pushed the manageable sizes of state spaces to $10^{20}$ or more states [34]. A BDD is a compact representation of a full Boolean decision tree over $K$ variables, obtained by merging common subtrees. In particular, a BDD is ordered if the variables labeling the nodes on any path from the root to a leaf respect the order $(x_K, \ldots, x_1)$, and is quasi-reduced if it contains no duplicate nodes. Note that a BDD can be exponentially more compact than its corresponding decision tree, but the degree of
compactness depends on the variable ordering chosen; unfortunately, finding an “optimal” ordering is an NP–complete problem [21]. The most commonly employed type of BDD is the reduced ordered BDD (ROBDD) [27, 28], where redundant nodes having identical children are eliminated. However, both BDDs and ROBDDs are canonical representations of Boolean functions).

The efficient implementation of BDDs requires to essential hash tables: a unique table to store and retrieve BDD nodes, and a cache to eliminate the need of evaluating a Boolean operation on a given combination of nodes more than once. Several C or C++ manipulation packages for ROBDD are available [18, 22, 93, 137, 139, 149]. Their efficiency has traditionally been measured by applying them to well–established circuit benchmarks, such as ISCAS85 [25] and ISCAS89 [24]; we refer the reader to [146, 161] for extensive performance studies of some of these packages. Probably the most important application areas of BDDs are state–space generation and model–checking algorithms [5, 29, 34, 62, 82, 112]. Various commercial tools have been built using the experience gained with academic model–checking tools [59, 63, 111, 112], and BDD–based techniques have begun attracting industrial attention [79, 140] for hardware verification. Symbolic model checking has also been investigated for analyzing software, often specified with Petri nets [20, 128, 129, 143, 165]. However, the resulting state spaces are usually too large to be represented using BDDs. One reason is that the behavior of software cannot be encoded as naturally in BDDs as that of digital circuits.

Several approaches have been studied to further improve the efficiency of BDD–based algorithms. Regarding time efficiency, breadth–first BDD–manipulation algorithms [10, 125, 126] have been explored and compared against the traditional depth–first ones. However, the results show no significant speed–ups, although breadth–first algorithms lead to more regular access patterns of hash tables and caches. Regarding space efficiency, a fair amount of work has concentrated on choosing appropriate variable orderings and for re–ordering variables on–the–fly [21, 68, 73, 74, 94, 135, 144, 158, 164]. Other sophisticated techniques exploit the component–based structure inherent in many digital systems, including partial model checking [7], transition–relation partitioning [33], and conjunctive decompositions [113]. Moreover, variants of BDDs able to support reasoning about aspects of real–time [85, 145, 163] and probability [13, 118] have been suggested. Finally, we point out that a few approaches to symbolic model checking have been developed, which do not rely on decision diagrams but instead on arithmetic and algebraic representations [11, 19, 20, 32].

### 2.3 Parallel Techniques

Both explicit and symbolic parallel state–space generation methods have been actively researched. Explicit techniques partition state spaces over multiple processors, thereby improving execution time and exploiting the larger overall main memory available on parallel machines. Many distributed–memory implementations reported in the literature (e.g. [152]) store states in a two–level hash table, where the use of a universal hashing function determining the processor “owning” a state ensures a randomized load balance. The reported speed–up is almost linear on a 32 node network of workstations (NOW) with a fast connection (Myrinet). In [35] state spaces are accessed via a hashing function defined on selected state components, in such a way that most transitions are between states owned by the same processor. The resulting speed–up is about linear for large enough problems, both on an IBM SP–2 and on a network of SUN Sparc workstations connected via 10Mb Ethernet. A different approach is used in [124], where states are stored in a balanced search tree. The first few levels of the tree are duplicated on each processor, while the remaining subtrees are mapped to individual processors. Dynamic load balancing is achieved by reassigning entire subtrees to different processors. Also shared–memory algorithms have been proposed for explicit state–space generation. For example, in [2] state spaces are stored in a B–tree, and an advanced locking mechanism on the tree nodes is used to reduce processor idle time and achieve good speed–ups.

Symbolic techniques share many issues with explicit techniques, such as load balancing, communication overhead, and termination detection. In addition, the parallelization of the algorithms is a problem in itself since
the traditional depth–first recursive BDD algorithms are inherently sequential [166]. Indeed, previous work on
distributed–memory BDD algorithms stresses the ability of using the overall amount of memory on a NOW, but do not achieve meaningful speed–ups compared to single–processor implementations, unless the latter start using virtual memory. In [138], BDD nodes are considered in a breadth–first order to improve memory access patterns and to reduce communication. However, four–processor execution is up to three times slower than for a single processor. Similarly, [155, 156] show speed–ups only when the sequential execution cannot fit in the main memory. Even then, the reported results refer to a specialized Meiko CS–2 machine. In [9], “software pointers” are used for reaching beyond the $2^{32}$ address–space limitation and solving the problem of pointer meaning across different processors, but the approach shows no speed–up at all. In [115], a small speed increase is achieved for ISCAS85 circuits [25]; the solution time on eight processors is 7% faster at best, but 176% slower at worst, when compared to the same code on a single processor. Only on an ad–hoc example, where the BDD has an exponential size in the number of variables, the speed–up reached 20%.

Better speed–ups have instead been demonstrated for shared–memory BDD algorithms. In [97] a speed–up factor of 10 is achieved using 15 of the 16 processors of an Encore Multimax. The algorithm employs the traditional BDD top–down recursion followed by bottom–up reduction. In [162] a hybrid approach with a limited breadth–first expansion is applied. More precisely, when too many nodes of a given level are in the work queue, the algorithm switches to depth–first, which has much smaller memory overhead. Speed–ups up to a factor of four are achieved on an eight–processor SGI Power Challenge with 1 GB of shared memory.

3 Proposed Work

The advent of BDDs increased the size of manageable state spaces by orders of magnitude and paved the road to successful industrial applications of symbolic model checking for hardware and synchronous software. However, asynchronous software systems pose new challenges for symbolic state–exploration algorithms, for the following reasons:

• **Space limitations.** State spaces of real–world distributed and embedded software systems are often beyond the limits of ordinary workstations. This is even true for BDD–based representations, where the memory requirements are linear in the number of BDD nodes, not in the number of encoded states. As the number of nodes varies during state–space generation, the peak BDD size is a critical factor. On a workstation with 1GB RAM, only up to 30 million nodes and their relative cache entries can be stored without making use of virtual or secondary memory.

• **Time limitations.** The generation of huge state spaces on a state–of–the–art workstation often takes hours or even days. This is unacceptable to software engineers who are interested in obtaining fast feedback from automated verification tools.

In contrast to all related work, we suggest a fundamentally novel way to address these space and time limitations when studying asynchronous software systems. We employ Multi–valued Decision Diagrams (MDDs) [95] for encoding state spaces, paired with a Kronecker encoding of the next–state function, which is inspired by techniques developed for the description and solution of large Markov chains [44, 31, 133, 153]. While being a relatively straightforward extension of BDDs, MDDs are essential to enable our Kronecker representation, which, in turn, results in completely different symbolic algorithms that exploit the **structural** [45] properties of the system under study and greatly improve space and time requirements.

The key effect of our structural approach is to enable a much finer manipulation of MDD nodes. With almost surgical precision, we can forecast and apply the effect of each event in an asynchronous system on portions of the MDD describing its state space, instead of using blunt monolithic applications of the next-state function.
encoded as a huge MDD, as traditional approaches do. This will lead to the development of sequential algorithms for MDD manipulation and state-space generation, as well as model checkers working on top of these, that are significantly more space and time efficient, as illustrated by some of our preliminary results which we will discuss in the next section. The capability of performing localized MDD manipulations will also enable the development of parallel symbolic state-space generation and model-checking algorithms along lines that are very different from those explored in the literature, promising significant speed-ups for both shared-memory and distributed-memory implementations.

3.1 Sequential MDD-based State-space Generation and Model Checking

In the following we address the most important novelties for the proposed sequential algorithms: the use of multi-valued decision diagrams and Boolean Kronecker encodings, event locality, and iteration control.

Multi-valued decision diagrams and Kronecker encodings. Given a discrete-state system composed of $K$ subsystems, a global state can be represented as a $K$-tuple of local states. Assuming finite state spaces, we can enumerate each local state space $S_k$ using traditional state-space generation techniques and identify it with the range $\{0, \ldots, |S_k| - 1\} = \mathbb{Z}$ of integers, for $K \geq k \geq 1$. Hence, a global state corresponds to a $K$-tuple of natural numbers, and a set of states $S$ can be represented by a characteristic function mapping the set $S_k \times \cdots \times S_1$ to $\{0, 1\}$, which in turn can be encoded by an MDD [95]. In particular, we use quasi-reduced MDDs, an example of which is illustrated in Figure 1: An internal node at level $k$ is depicted as an array of $|S_k|$ arcs to nodes at level $k - 1$, and state $(i_k, \ldots, i_1)$ belongs to the set encoded by the MDD if and only if, starting at the root and following the $i_k$-th arc when in a node at level $k$, we reach terminal node 1.

The next-state function $N$ is usually encoded as a $2K$-level decision diagram over $(x_K, y_K, \ldots, x_1, y_1)$, where $x_k$ and $y_k$ refer to the $k$-th state component before and after the step, respectively. Figure 2 illustrates instead our approach on an example Petri net [120, 141] whose places have been partitioned into $\{p\}$, $\{q, r\}$, $\{s\}$, and $\{t\}$, defining four corresponding subnets. Central to our idea is the automatic extraction of structural information from such a high-level model. First of all, a disjunctively-partitioned next-state function [33] has long been considered effective for asynchronous systems. In our structural approach, this means recognizing a set $E$ of possible events so that we can write $N = \bigcup_{\alpha \in E} N_\alpha$, where $N_\alpha(i_k, \ldots, i_1)$ describes the states reachable from state $(i_k, \ldots, i_1)$ when event $\alpha$ occurs. We use a further decomposition along a different axis, by expressing the effect of an event on each submodel (level of the MDD). For example, the occurrence of event/Petri net transition $d$ changes the local state of the two subnets defined by $\{s\}$ and $\{t\}$ and no other local states. This is reflected by the “level×event” table shown on the right of Figure 2: the top two entries in the column for event $d$ are empty, i.e., they are identity matrices. The result is an extremely compact representation of $N$ as the Boolean Kronecker expression $\Sigma_{\alpha \in E} \bigotimes_{K \geq k \geq 1} N_\alpha \equiv \bigwedge_{\alpha \in E} \exists k \in \{K, \ldots, 1\}, N_\alpha[i_k, j_1] = 1$. 

Figure 1: A four-level MDD on $\{0,1,2,3\} \times \{0,1,2\} \times \{0,1\} \times \{0,1,2\}$ and the set $S$ it encodes.
Our preliminary results with using MDDs to store the state space and Kronecker encodings to store the next-state function, reported in [117], were extremely encouraging, but also raised several important questions that need to be addressed:

• **How to obtain a good partition?** The only requirement on the partition defining the subsystems, each of which is then mapped to a different MDD level, is that it is consistent [45] with the Boolean Kronecker expression of $\mathcal{N}$. We originally called this property “being in (logical) product-form” [44], since it has strong similarities with the analogous one for product-form queuing networks [14], which deals, however, with real-valued quantities. In our tool SMART [39], the system is specified as a Petri net and it is possible to automatically derive the finest consistent partition or, alternatively, to check that a user-specified partition is consistent. However, we have shown in [117] that the partition choice greatly affects the efficiency of the computation. While it is clear that the reduced height of MDDs vs. BDDs, which is due to the use of integer variables instead of binary variables can be an advantage, we also experienced that too few levels with very large nodes cannot be stored or handled efficiently. Thus, we propose to investigate heuristics to coarsen the finest partition with the goal of reducing space and time requirements. We expect structural model information, e.g., place-invariants in the case of Petri nets [120], to provide the basis for such heuristics.

• **How do partitioning and variable ordering interrelate?** It is well known that variable ordering affects the sizes of BDDs and MDDs. We expect heuristics derived for model partitioning to be strongly influenced by the order in which the $K$ subsystems are considered. Indeed, it is likely that good heuristics will have to consider both variable ordering and partition ordering. In comparison to BDDs where only the variable ordering needs to be chosen, our approach certainly provides greater flexibility, but also presents a more challenging optimization problem.

• **How can MDD nodes be shared efficiently?** While state-space generation builds a single MDD, model checking operates on multiple sets of states, each encoded as an MDD on the same set of $K$ variables. For efficiency, nodes must be shared between different MDDs, just as in BDD approaches [12, 116, 135, 157, 158]. Using MDDs instead of BDDs does not complicate matters in this respect. However, in conjunction with the advanced ideas we discuss next, where MDDs are not necessarily traversed from top to bottom, keeping track of nodes belonging to different MDDs requires more complex solutions.
**Event locality.** Key improvements we propose emerge from the inherent event locality [40] of asynchronous event-driven systems. In such systems, the occurrence of an event changes only some components of the global state vector. Due to our mapping of subsystems to MDD levels, we can exploit event locality to explore MDDs only between the maximum level $\text{Top}(\alpha)$ and the minimum level $\text{Bot}(\alpha)$ affected by each event $\alpha$. This range of levels (see Figure 2) can be extracted automatically from the high-level description of the system prior to state-space generation. To benefit from this observation, we access MDD nodes by “jumping in the middle” of MDDs, to level $\text{Top}(\alpha)$, rather than always starting MDD operations at the root, as is done by traditional approaches. Moreover, operations on MDDs — such as those corresponding to the union or intersection of the encoded state spaces — can update nodes in-place rather than returning their results by creating new nodes, as is done in previous works. These observations led us to the prototypic algorithms reported in [38], where we demonstrated up to one of order of magnitude speed-ups over those in [117]. Furthermore, the improvement grows with the height of MDDs. Accessing nodes in the middle and updating them in place, however, means that they may become non-unique and have to be deleted. We then need to notify upstream nodes, i.e., nodes towards the roots, of this change, while traditional BDD approaches only have to follow links downstream, i.e., towards the leaves. In this research context, several issues remain open:

- **What are the design choices regarding level-wise data structures?** Most of a memory’s node stores the arcs to its descendants. Exploiting locality implies that we need to store, organize, and provide access to nodes by levels. Our adoption of quasi-reduced MDDs already helps this task: forcing arcs to span only one level simplifies and reduces node storage, possibly more than offsetting the space occupied by redundant nodes. We need experimental data to establish whether this is a good trade-off. Also, regarding the upstream propagation of work, several solutions should be explored.

- **Can we exploit locality to reduce cache memory requirements?** For state-space generation, when exploring event $\alpha$ in a node at level $\text{Top}(\alpha)$, our algorithms access that node once, directly. Traditional approaches instead access that node once for each of its incoming arcs, and use the operation cache to make all but the first access inexpensive. Hence, we can save memory by storing in the cache the result of exploring $\alpha$ only in nodes at levels below $\text{Top}(\alpha)$ and up to $\text{Bot}(\alpha)$, but not at level $\text{Top}(\alpha)$ itself. Additional ways to save on cache entries along these lines should be explored.

- **Should the partitioning and variable-ordering heuristics favor a high degree of event locality?** The partition chosen for the model determines the levels affected by an event $\alpha$, while the submodel order determines the distance between $\text{Top}(\alpha)$ and $\text{Bot}(\alpha)$. Execution time improves when most events interest a small range of levels; thus, our partitioning and ordering heuristics should try to minimize, for example, $\sum_{\alpha \in E}(\text{Top}(\alpha) - \text{Bot}(\alpha))$. Figure 2 illustrates another interaction between locality and partitioning: observe that we have merged events $b$ and $c$ into a single macro-event $\{b, c\}$, since both affect only level 3 of the MDD. Reducing the number of events that need to be considered increases the efficiency of state-space generation, and, in the example, this would not be possible if we would assign places $q$ and $r$ to different subnets. However, we must remember that the main goal of the partitioning and ordering heuristics is to reduce the size of the generated MDDs.

**Iteration control.** Another aspect we intend to explore in conjunction with event locality concerns the order in which MDD nodes and system events are considered by fixed-point operations such as those needed for state-space generation or symbolic model checking [56, 112]. In our initial experiments with state-space generation [38], we partitioned the events $E$ into $K$ classes, $E_k, \ldots, E_1$, where $E_k$ containing all events affecting level $k$ and possibly lower levels, i.e., all $\alpha$ for which $\text{Top}(\alpha) = k$. Then, we explored the MDD nodes not in depth-first or breadth-first fashion starting at the root, but by levels. More precisely, we directly access all nodes at the bottom level, 1, and apply the next-state function for the events in $E_1$. Then we consider all the nodes at level 2,
and so on, up to the nodes at the top level, $K$. This completes one iteration. The intuition behind this approach is to find as many new states as soon as we can. This simple idea to guide event exploration works very well, and it can even reduce the number of iterations required for convergence: by exploring events affecting higher levels after those affecting lower levels, we effectively increase the likelihood of performing multiple next-state function steps in a single iteration. For example, employing this idea, the classic model of the dining philosophers and that of a slotted ring network converge in two and $N/2 + 2$ iterations, respectively, instead of the $2N + 1$ and $N^2 + 16$ iterations required by the standard BDD approach used in [129]; here, $N$ is a model parameter affecting the implied state-space sizes.

However, we achieved by far the largest impact along these lines using the idea of saturation [46]. Given the MDD encoding the initial system states, we consider each node at level $k$, starting from level 1 and moving up, and repeatedly apply all events in $E_k$ to it until no more states are found, i.e., until the set encoded by the node has reached a fixed point with respect to $E_k$. This saturation can cause the creation of new nodes at lower levels, which are then recursively saturated right away before completing the saturation of the nodes above them. The resulting algorithm has many desirable properties: the unique table needs to store only the saturated nodes, and these are likely to remain useful throughout the execution (any unsaturated node placed in the unique table by traditional algorithms is instead guaranteed to be eventually deleted); the operation caches are used in a much more localized way, thus their memory requirements are greatly reduced; finally, an event is explored on nodes whose descendant are saturated, thus we again increase the likelihood of finding new states early. Saturation is quite innovative in that it has no concept of global fixed–point iteration: state–space generation ends when the root MDD node is reached and saturated.

Each of our improvements (MDDs, Kronecker encodings, locality, and saturation) has a cumulative effect. Table 1 compares their preliminary implementation in our tool SMART with the widely–known model checker SMV, or to be precise, its public–domain incarnation NuSMV [50] that is implemented around optimized, standard BDD–based techniques. In addition to the dining philosophers model from the NuSMV distribution, we consider models of a slotted ring network [129], a round–robin mutual exclusion protocol [81], and a flexible manufacturing system [49], all of them parameterized by an integer $N$ affecting the state–space sizes. For each model we report the final and peak memory (in kilobytes) required to store the state space and the runtime (in seconds), for a small value of $N$ and for the largest value of $N$ that NuSMV or SMART can comfortably solve. The table clearly shows that our algorithm outperforms classical algorithms by up to four orders of magnitude in time and over three orders of magnitude in memory, and the gap keeps increasing as models grow.

Model checking. The abovementioned very encouraging results in state–space generation prompt us to adapt our ideas to the more general setting of symbolic model checking, for which the most challenging question is the following:

- **Can we use iteration control strategies for model–checking?** We have clearly shown that saturation excels at state–space generation, which is essentially analogous to the $EF$ operator in CTL. The $EG$ operator, instead, requires to restrict the exploration only to paths along which a certain property $q$ holds at all times. This is traditionally done by alternating one step of the previous–state function and one intersection with the set of states where $q$ holds. This would apparently preclude the possibility of applying saturation, which has no concept of “step.” However, a structural analysis of the high–level model can help, since it allows us to determine which events may affect predicate $q$. We can then apply saturation only for those that do not affect $q$, while using the traditional approach for the ones that do. Analogously, more efficient fairness checks might be possible as well by using saturation–based iterations that isolate events affecting (or not

---

1 In our setting, the previous–state function is simply obtained by transposing our Kronecker matrices encoding the next–state function.
<table>
<thead>
<tr>
<th>N</th>
<th>Reachable states</th>
<th>Final memory (kB)</th>
<th>Peak memory (kB)</th>
<th>Time (sec, 800MHz P-III)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SMART</td>
<td>NuSMV</td>
<td>SMART</td>
</tr>
<tr>
<td><strong>Dining Philosophers (N MDD levels)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>$2.23 \times 10^{31}$</td>
<td>18</td>
<td>10,800</td>
<td>22</td>
</tr>
<tr>
<td>200</td>
<td>$2.47 \times 10^{125}$</td>
<td>74</td>
<td>27,155</td>
<td>93</td>
</tr>
<tr>
<td>10,000</td>
<td>$4.26 \times 10^{6269}$</td>
<td>3,749</td>
<td>—</td>
<td>4,686</td>
</tr>
<tr>
<td><strong>Slotted Ring Network (N MDD levels)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$8.29 \times 10^{9}$</td>
<td>4</td>
<td>5,287</td>
<td>28</td>
</tr>
<tr>
<td>15</td>
<td>$1.46 \times 10^{15}$</td>
<td>10</td>
<td>9,386</td>
<td>80</td>
</tr>
<tr>
<td>200</td>
<td>$8.38 \times 10^{211}$</td>
<td>1,729</td>
<td>—</td>
<td>120,316</td>
</tr>
<tr>
<td><strong>Round Robin Mutual Exclusion (N+1 MDD levels)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>$4.72 \times 10^{7}$</td>
<td>18</td>
<td>7,300</td>
<td>20</td>
</tr>
<tr>
<td>100</td>
<td>$2.85 \times 10^{32}$</td>
<td>356</td>
<td>16,228</td>
<td>372</td>
</tr>
<tr>
<td>300</td>
<td>$1.37 \times 10^{93}$</td>
<td>3,063</td>
<td>—</td>
<td>3,109</td>
</tr>
<tr>
<td><strong>Flexible Manufacturing System (19 MDD levels)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$4.28 \times 10^{6}$</td>
<td>16</td>
<td>1,707</td>
<td>26</td>
</tr>
<tr>
<td>20</td>
<td>$3.84 \times 10^{9}$</td>
<td>55</td>
<td>14,077</td>
<td>101</td>
</tr>
<tr>
<td>250</td>
<td>$3.47 \times 10^{26}$</td>
<td>25,507</td>
<td>—</td>
<td>69,087</td>
</tr>
</tbody>
</table>

Table 1: Time and memory requirements: SMART (MDD/Kronecker–based) vs. NuSMV (BDD–based).

Due to the subtleties of the many issues involved in devising an efficient sequential model checker, we are planning to formally proof its correctness with the help of a theorem prover, such as PVS [127]. This requires building a library for reasoning about shared MDDs and devising proof strategies/tactics that can discharge proofs of simple statements automatically. While this requires some effort that will largely be put–in by the international collaborator, it will almost certainly permit one useful insights into the properties of our model–checking algorithm, which might in turn be exploited to improve the algorithm further.

### 3.2 Shared–memory and Distributed–memory Algorithms

Especially when studying distributed software, the sizes of MDDs grow very rapidly. Hence, parallel algorithms for MDD–manipulations might be the answer for providing results in a timely manner. Unlike previous approaches to parallelize BDD operations mentioned in the background section, we intend to find **parallelism at the event level** by exploiting event locality. For distributed–memory approaches, this will not only allow us to utilize the significantly larger main memory available on PC–clusters and networks of workstations, but also provide the much needed speed–up that has eluded researchers so far.

**Parallelization for shared–memory architectures.** For the design of shared–memory algorithms we will focus on investigating the following major issues:

- **What sources of parallelism should be exploited?** Our sequential approach lends itself to both event–based parallelism, where each event is explored by a single thread over entire MDDs, and level–based parallelism, where all events are explored by a single thread over a given level. It is possible that the best option will be a mixture of the two. Neither kind of parallelization has been considered for symbolic state–space generation or symbolic model–checking before.
• **How to maintain cache consistency?** Regardless of the parallelization choice, it is certain that the operation caches will be examined by many threads. Since cache look–up is an extremely frequent operation, it will be imperative to use cache access protocols which do not result in large run–time penalties. Hence, such protocols must either avoid the use of locks or use them very progressively.

• **How to minimize idle time?** With event–based parallelism, threads will require locks on individual MDD nodes to ensure exclusive access. The algorithms to be developed must exploit the large number of MDD nodes to their advantage, reducing the likelihood of threads having to wait at locked nodes.

Our shared algorithms will be initially developed on the four-processor Sun workstations available in the PI’s department, under Pthreads [103]; if needed, the PI will secure access to multiprocessors with larger number of nodes. To assess the effectiveness of our parallelizations in terms of run–time results, these algorithms will be applied to the same case studies as the sequential algorithms. In addition, the algorithms will also be formally verified.

**Parallelization for distributed–memory architectures.** For distributed–memory architectures, the algorithms we intend to develop will focus on two important goals, namely achieving parallelism and reducing communication overhead. Event locality will play a key role for achieving both goals.

• **How to achieve high, scalable levels of parallelism?** Assuming that the number $K$ of levels is larger than the number $N$ of processors, we can partition the levels into $N$ contiguous ranges and assign each range to a different processor. Then, event locality will allow us to trigger event exploration at the required local levels and events in different ranges will be naturally explored in parallel, without the obvious bottleneck of having to start all work at the top level of MDDs. Alternatively, we will explore the allocation of individual nodes to processors, with the goal of keeping the range of most events within a single processor, as much as possible. However, this will imply more complex pointer management, and the run–time costs of this choice should be carefully evaluated against the increased flexibility. Since many of the allocation heuristics will be based on purely structural considerations, they can be taken into account in a preliminary analysis step, prior to any fixed–point iteration.

• **How to achieve memory and load balance?** With a partition of levels over processors, memory and load unbalance can be corrected by shifting the border between ranges. With the finer granularity achieved by assigning nodes to processors, balancing should be easier. Another issue to consider when partitioning the levels or nodes over the available processors will be the number of arcs connecting nodes on different processors. This is analogous to a problem we discussed in [35] for explicit state–space generation, although assessing the effect of different design decisions is much harder in the symbolic case we intend to tackle.

• **How to best manage message passing?** In most MDD manipulations, work requests will be propagated towards the MDD leaves, while the results will be propagated towards the MDD roots. According to our strategies for parallelization, this requires to pass messages between different processors. While the frequently used technique of batching messages allows for lowering communication overhead, it will have to be carefully tuned to limit the danger of increasing processor idle time.

• **What procedure should be used to detect termination?** In [35], we used a barrier synchronization [122] for detecting termination. However, many options for termination detection exist in a distributed environment; see [110] for an extensive taxonomy of such algorithms. The best algorithm to fit our needs will have to be evaluated carefully.
We plan to implement the distributed algorithms using MPI (either MPICH or LAM) [147] and run them on Beowulf–style PC–clusters [151]. One such large cluster is already available to the PI, whose department is a member of the College of William and Mary Computational Science Cluster. As with our previous algorithms, the distributed ones will also formally proved analyzed using a theorem prover, to ensure their correct operation.

3.3 Tool Integration, Benchmarking, and Case Studies

To evaluate our research, we will first employ our sequential and parallel MDD–manipulation algorithms to develop and implement model checkers for the temporal logic CTL [52] within our tool SMART [39]. The utility of our algorithms are then evaluated in two ways. First, their performance will be analyzed using a novel benchmark that we are planning to devise. Second, we will conduct two case studies which promise to be particularly challenging. One case study deals with the verification of new access protocols in distributed B–tree algorithms. The other case study concerns the application of formal verification to investigate issues of human–computer interaction in avionics systems, a problem in which our international collaborator has expertise. Together, the case studies are meant to be illustrative of the application areas targeted by the proposed work.

Tool integration. The algorithms we are going to develop will be implemented in SMART, a modeling tool for the logical and timing analysis of complex systems, which is developed under direction of the PI at the College of William and Mary and is written in C++.

Interfacing and input languages. The architecture of SMART has a clear separation between the front–end (user–visible interface, languages, and high–level formalisms) and the back–end (state–space generators, Markov solvers, and discrete–event simulators). This means that inserting new “solution” modules, such as model checkers, can be done with relatively little effort, as most of the remaining code is not affected by such modifications. As a consequence, any high–level formalism, as well as any other solution module requiring state–space generation as a preliminary step, can immediately make use of the new techniques we will provide. For the type of applications we envision, one very appropriate language front–end, namely Petri nets [120, 141], is already fully implemented. It is currently being extended by constructs that allow for an easier component–based specification of systems. This will in particular support elegant specifications of distributed software systems which are considered in this proposal. As previous work has shown, it is also easy to integrate MDD-based manipulation routines [38, 46, 47, 117] into SMART.

Symbolic model checking. SMART supports the description of “atomic propositions” as expressions relating the number of tokens in the places of a Petri net. Then, building on these propositions, SMART provides the functionality of a basic CTL [52] model checker. However, much work still needs to be done in this area, both to provide witnesses and counter–examples, and to increase its efficiency to the same level of our state–space generation algorithms: our goal is to remove one of the qualitative differences between symbolic state–space generation and symbolic model checking mentioned in [161]. We plan to build a full–strength symbolic CTL model checker on top of our sequential and parallel MDD packages, as well as a better interface for inputting CTL formulas into SMART and an output facility for complex witness or counter–example paths. With the efficient underlying MDD–manipulation routines we envision and our advanced iteration control, we expect SMART to be very time and space efficient when verifying asynchronous software. It should also be mentioned that our implementation in form of C++ packages will allow an easy integration and comparison with existing verification and validation tools other than SMART, as well as with industrial design tools.

Benchmarking. As indicated earlier, the performance of symbolic model checkers is usually measured by applying them to the well–established ISCAS85 [25] and ISCAS89 [24] benchmarks that include hardware example systems taken from circuit design. These systems have a synchronous semantics and thus the ISCAS benchmarks are not applicable to our model-checking algorithms, which are after all targeted towards verifying asynchronous
software systems. Indeed, the current literature does not provide any suitable benchmark for such asynchronous systems, with many papers just briefly studying one or two small example systems each. In this light, we will devise a new benchmark by systematically collecting a variety of asynchronous example systems that have been studied by those software engineers who are interested in the application of formal methods, as well as other example systems analyzed by concurrency theoreticians. This benchmark will be well-documented and made available to the wider research community via the Internet. Of course, it will be used for evaluating the performance of the algorithms devised within this project. In addition, our algorithms will be made web-accessible for remote execution at the College of William and Mary. This allows for gathering run-time statistics that will enable to fine-tune the algorithms’ performance with increasing experience.

**Case study: Parallel algorithm verification.** The first case study focuses on a shared-memory algorithm for parallel B-tree manipulation, currently being investigated by the PI [123]. B-trees [15] are a popular data structure to index very large data sets frequently used in database applications including distributed databases for banking and reservation systems. The need to allow concurrent access to B-trees, e.g., for distributed databases, was already recognized in [16], where a seminal mechanism based on various levels of node locks is described. Reader and writer processes descend the B-tree from its root, locking nodes before accessing their keys and contents. A read lock on a node allows other readers, but not writers, to access it, while a write lock prohibits other processes to access the node. However, locks introduce severe performance problems, both because they reduce parallelism and because accessing a lock requires a substantial number of processor cycles on some parallel machines. The first problem is especially important with traditional concurrent access algorithms [16], where a process starts locking the root of the B-tree and continues locking nodes down on a path towards the leaves, unlocking them only when safe. For example, a writer unlocks a node only if its (locked) descendant on the path is not full. This means that the first few levels of B-trees often contain several locked nodes and reduce the potential parallelism.

The approach we are investigating uses speculative computing in a B-tree environment with batched deletion, i.e., the set of keys is never decreased during concurrent operations. Readers do not lock nodes at all, but simply record the modification count of the nodes encountered on their search path. If they find the sought key, they are assured of correct operation. Otherwise, they check the current modification counts of the nodes on the path against the recorded ones; if they match, they are again assured of correct operation, else they abort and restart the read. Such an approach can greatly increase parallelism if read operations constitute a majority of the accesses. Furthermore, it can speed up the execution if locks are expensive to acquire, since reads need no locks at all.

The correctness of our speculative approach, however, is quite hard to establish, as all possible interleavings of multiple writer and reader executions must be taken into account. Traditional testing is inadequate to discover possible errors due to infrequent race conditions. We strongly believe that our parallel MDD-based state-exploration techniques will enable the investigation of the algorithm for realistic operating environments. In particular, we plan to model check whether the algorithm’s potential executions do not introduce any inconsistencies in B-tree nodes (a safety property) and that every reader and writer is eventually permitted access (a liveness property). This will testify to the use of model checking in engineering practice.

**Case study: Analyzing mode confusion.** Deficiencies in human–computer interaction are increasingly contributing to incidents and accidents in air traffic [90]. Recent advancements in digital flight decks pose several challenges to aircraft pilots. As an example, consider flight guidance systems that are part of airborne flight control systems. Flight guidance systems continuously determine the difference between the actual state of an aircraft — its position, speed, and attitude as measured by its sensors — and its desired state as indicated by the crew or via the flight management system. In response, the system generates commands to minimize this difference, which the autopilot may then translate into movements of the aircraft’s actuators. The complexity of the
underlying algorithms may, in situations of high workload, confuse pilots about the actual state in which the flight guidance system is. In human–factors research, this kind of confusion is referred to as mode confusion [102].

Our international collaborator has been involved with preliminary research aimed at identifying potential sources of mode confusion, including the formal modeling and verification of a few simple, mostly synchronous, components of an idealized flight guidance system [114]. These were modeled as finite state machines, and the mode–confusion properties of interest were specified as temporal–logic formulas. Investigated properties included (i) inconsistent behaviors, e.g., whether a switch or dial has different functionality in different system states, (ii) ignored operator inputs, e.g., whether a crew input does not eventually result in the desired change of state, and (iii) indirect mode changes, e.g., whether the system changes its state without crew input. Conventional symbolic model checking techniques turned out to be well suited for checking these properties [107, 121]. The obtained results were very well received by avionics researchers and encourage us to continue the chosen path of employing symbolic model checking to analyze mode confusion.

In the proposed study, we plan to investigate new mode confusion properties involving the gathering and display of mode information in the cockpit, by modeling large parts of real–world flight decks. In fact, many properties of interest can only be thoroughly analyzed when also several parts of the operating environment of flight guidance and flight control systems, such as switching panels, displays, and sensor and actuator behavior, are taken into account. This will lead to distributed asynchronous models with enormous state spaces that cannot be handled by existing symbolic state–space exploration techniques — at least not if results are to be returned within seconds, as hoped by human–factors researchers. Hence, this study will be an excellent candidate for our novel parallel symbolic model–checking algorithms. In addition, we also expect the results to be of great interest to the avionics community.

3.4 Schedule and Distribution of Work

The first year of the project will concentrate on developing and implementing sequential algorithms for MDD–based state–space generation in which the PI, the co–PI, and the two graduate students will participate. The PI and one student will focus on the algorithms’ design, while the co–PI and the other student will address their proof of correctness, focusing in particular on proving correct cache management in conjunction with our optimizations. The integration of a model–checking front–end in the Petri net tool SMART [39] will also be addressed in the first year. Our sequential algorithms will be fine–tuned in the second year and made web–accessible. Performance results will be collected by applying the algorithms to standard benchmark applications introduced in the literature as well as to our two proposed case studies.

Work on both shared–memory and distributed–memory algorithms will begin in the second year. The PI and the co–PI will oversee the designs regarding both directions, and each graduate student will specialize in one of them. The aspects of exploiting event–locality and iteration control, which are already addressed in the first year, will continue to be investigated in the second year, as new issues are likely to emerge when parallelizing our algorithms. As with the sequential algorithms, the parallel algorithms will be implemented by the graduate students as soon as they are designed. By the beginning of the third year, the parallel algorithms will be made available online. The third year will see a substantial implementation and experimentation effort on both shared–memory and distributed–memory algorithms. The last six months of the project will be dedicated to fine–tuning these algorithms, to gather run–time statistics on execution and memory behavior, and to assess the speed–ups of the shared–memory and distributed–memory implementations.

Work on the proposed case studies will be done throughout the duration of the project. While the PI and one graduate student will concentrate on verifying the concurrent B–tree algorithm, the co–PI together with the other student will address the analysis of mode–confusion.
3.5 Synergy between Research and Education

The proposed research focuses on parallel and distributed algorithms for decision diagrams, targeting the application area of software verification. Knowledge of parallel and distributed algorithms and the ability to implement them is an essential skill that all graduate students must acquire, and most of them are very much interested in doing so. Decision diagrams are a data structure that should ideally be mastered by any graduate student as well. Unfortunately, decision diagrams are often ignored in a standard curriculum, no doubt due to their relatively recent emergence as a useful and important data structure. The project we propose will provide ample research opportunities for the graduate students involved, allowing them to learn about both parallel and distributed algorithms and decision diagrams. It is certain that the project will also provide material for several PhD dissertations.

The investigators currently teach two courses related to the proposed research in the Department of Computer Science at the College of William and Mary. The PI teaches the advanced graduate course CSci 746: Discrete-state Stochastic Models, where about 25% of the lectures are concerned with the concept of state space and transition graph as well as with efficient techniques for their management. The co-PI, who has an adjunct assistant professor appointment in the department, teaches the course CSci 780: Program Verification, dedicated to the mathematical foundations of program verification and analysis.

A new graduate, seminar-style course on Decision Diagram Technologies is being planned by the investigators. This will be a further vehicle to attract graduate students to the proposed topic, in addition to the three students already working on state-space techniques with the investigators. The first one, A. Miner, is now completing his PhD work. A good portion of his thesis deals with decision diagrams and related structures, with a particular focus on stochastic modeling applications [40, 41, 117, 118]. The second student, R. Siminiceanu, is in his second year of the PhD program and has already co-authored a paper on improved decision diagram techniques [38]. This paper describes the results of work he performed together with the PI (his advisor) and the co-PI (the external member of his PhD committee) during a summer stay at ICASE, the employer of the co-PI. The third student, M. Griffith, is completing her MS project on partial state-space generation using probabilistic methods with the PI and has just been formally accepted in the PhD program.

The PI and co-PI strongly believe in the importance of implementing new research results in usable prototypes, with the goal of applying them to real-world problems. The tool SMART [40] is being successfully used at the College of William and Mary as the platform for such prototypes, providing a synergistic advantage for all involved students, so far one undergraduate, one MS, and four PhD students. Regarding applications, the investigators will target both government agencies and private industries, where the enhanced abilities provided by the proposed research will enable the supported students to become involved with real projects. One particular opportunity is in the area of aerospace applications, thanks to the investigators relations to NASA Langley Research Center and to the Virginia Space Grant Consortium.

4 Results from Prior NSF Support

Prof. Ciardo is the PI (with Prof. E. Smirni of William and Mary as co-PI) of NSF grant CCR-0098278, “Effective Techniques and Tools for Resource Management in Clustered Web Servers”, $279,485, 7/16/2001 - 7/15/2004. Currently, Wei (Helen) Sun, an MS student, and Alma Riska, a PhD student, are being supported by this project. In addition, Eric Davis, an undergraduate student, joined our group and is working on clustering techniques for web workload characterization for his Honor’s project. Even if the project just started recently, we already have exciting results. We have introduced EQUILOAD, a new parameterized size-based request allocation policy that can be used in clustered web servers to achieve both excellent memory locality and load balance. One publication supported by this grant is [48], which appeared in a special issue of the journal Performance Evaluation. A second publication, in which we develop a dynamic self-adjusting mechanism to adjust EQUILOAD’s parameters based
on the history of requests seen by the web server cluster, was just recently accepted to ICDCS (International Conference on Distributed Computing Systems) [142].

Prof. Ciardo was also the PI (with Prof. D. Nicol of Dartmouth College as co–PI) on the project “Integrated Modeling Project”, $36,500, 9/1/1995 - 8/31/1996, a subcontract from Duke University on NSF grant EEC-9418765. In the first year, the award supported a graduate student’s work on a prototype implementation of a new distributed state space generation algorithm for Markov models, which showed excellent speed–up [35, 124]. In the second year, a second graduate student was supported to work on discrete-event simulation of models with both discrete and continuous components [42] (subsequently selected to be extended and published in IEEE TSE [43]) and on non-Markov processes [36] (also in this case the paper was selected for publication in extended version, in Performance Evaluation [37]).
References Cited


Rationale for our international collaboration

Much of the inspiration for the proposed research can be traced back to a collaboration between the PI and Dr. Gerald Luettgen, beginning in Summer 1999. At that time, Dr. Luettgen was a Staff Scientist at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA, and the PI visited ICASE as a Summer Visitor.

The PI, whose original background is in Petri nets, stochastic Petri nets, numerical solution of Markov chains, and performance modeling, had begun working on Kronecker representations for large Markov chains and multi-valued decision diagrams for the efficient storage of their state spaces. Dr. Luettgen, whose background is instead in formal methods, semantics of specification and programming languages, had started applied research on model checking for problems of interest to NASA, such as mode confusion in cockpits.

During that summer the two, together with Radu Siminiceanu, a graduate student at William and Mary, began considering the applicability of MDD techniques to the general state-space generation and symbolic model-checking settings. Since then, they have collaborated on two fundamental papers, each of which has substantially advanced the efficiency of state-space generation. Although Dr. Luettgen is now a Senior Lecturer in Computer Science at the University of Sheffield, UK, the collaboration with the PI is ongoing. Dr. Luettgen is currently an external member of Mr. Siminiceanu’s PhD Committee and visited William and Mary in January 2002 for the Proposal Defense; conversely, Dr. Luettgen’s students are using the SMART tool, developed by the PI, in their research at the University of Sheffield.

By supporting this collaboration, the National Science Foundation will enable the ongoing continuation of this research and will increase the PI’s ability of reaching a wider audience for the results to be developed.
GIANFRANCO CIARDO
Associate Professor of Computer Science

Department of Computer Science   Phone:  (757) 221–3478
College of William and Mary   Fax:  (757) 221–1717
P.O. Box 8795   E-mail: ciardo@cs.wm.edu
Williamsburg, VA 23187–8795   URL:  www.cs.wm.edu/~ciardo


Professional preparation:

- **April 1989:** PhD in Computer Science, Duke University, Durham, NC.
- **July 1982:** Laurea *summa cum laude* in Computer Science, University of Torino, Italy.

Appointments:

- **August 1997 – present:** Associate Professor, **August 1992 – July 1997:** Assistant Professor, Department of Computer Science, College of William and Mary, Williamsburg, VA.
- **Spring 2000:** Sabbatical visit, HP Labs, Palo Alto, CA.
- **September 1988 – July 1992:** Member Tech. Staff, Software Productivity Consortium, Herndon, VA.

Selected awards:


Five publications most relevant to the project:


**Five other publications:**


**Synergistic activities:**

- **Associate Editor**, IEEE Transactions on Software Engineering (since February 2001).
- **Co–Organizer** of the Dagstuhl Seminar on Performance and Dependability Modelling with Stochastic Petri Nets, Saarbrücken, Germany, May 1995. *Visiting Professor* at Berlin University of Technology, Germany (Fall 1992) and University of Torino, Italy (Fall 1999). *Lecturer*, First Euro Summerschool on Trends in Computer Science: Formal Methods and Performance Analysis, July 2000, Nijmegen, the Netherlands.
- Leader of the SMART development team, a tool integrating high–level stochastic modeling formalisms and solution techniques. SMART was released in December, 2001.
- **General Chair**, ICATPN, June 1999, Williamsburg, VA. This prestigious conference devoted to theory and applications of Petri net has been held in the US only twice in its 21-year history.

**Collaborators:** P. Buchholz (Dresden Univ. of Techn., Germany), L. Cherkasova (HP Labs, Palo Alto, CA), S. Donatelli (Univ. Torino, Italy), R. German (Berlin Univ. of Techn., Germany), G. Hommel (Berlin Univ. of Techn., Germany), P. Kemper (Univ. of Dortmund, Germany), L. Leemis (College of William and Mary, Williamsburg, VA), C. Lindemann (Univ. of Dortmund, Germany), G. Lüttgen (Univ. of Sheffield, UK), J. Muppala (Hong Kong Univ. of Science and Techn., China), D. Nicol (Dartmouth College), M. Squillante (IBM, T.J. Watson Research Center, NY), W. Stewart (NC State Univ.), Y. Takahashi (Tokyo Inst. of Tech., Japan), M. Tilgnier (Tokyo Inst. of Tech., Japan), R. Zijal (Berlin Univ. of Techn., Germany).

**Past and present graduate students** (all at the College of William and Mary): H. Kim (Undergraduate Honors, 1994), C. Ji (MS, 1996), D. Galayda (MS, 1997), G. Li (MS, 1998), J. Maddalon (MS, 1999), S. Griffith (MS, 2000), A. Mangalam (MS, 2000), P. Sugden (MS, 1999), A. Miner (PhD, 2000), R. Jones (PhD, expected 2001), R. Siminiceanu (PhD, expected 2002), R. Marmorstein (MS, expected 2002). W. Sun (MS, expected 2002).

**Thesis advisor:** K.S. Trivedi (Duke University, Durham, NC).
Budget Justification

Faculty salary: One month of summer support with standard benefits (FICA at 7.65%) is requested for the PI for the first and second year of the contract, and two months for the third year.

Graduate students: Support is requested for two graduate students for each year of the project. The requested sum includes nine months half–time support during the academic year plus three months full–time support during the summer session. Standard benefits ($900 for health insurance, FICA at 7.65% of the summer salary) and the students’ tuition ($5,720 per student per year, under Other Direct Costs) are also requested.

Travel: Funds are requested to attend two domestic (assumed cost: $1500/trip) and two international (assumed cost $2,000/trip) conferences each year. Approximately half of the domestic travel support will be used for the graduate students involved in the project, in order to provide them with first–hand exposure to world–class researchers. The conferences of interest include those traditionally dedicated to formal verification, such as the ACM/IEEE Design Automation Conference, the IEEE International Conference on Computer–Aided Design, the International Conference on Formal Methods in Computer–Aided Design (FMCAD), the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), the International Conference on Computer Aided Verification (CAV), and the International Conference on Foundations of Tools and Algorithms for the Construction and Analysis of Systems (TACAS). However, we also intend to present our results at broad conferences on software engineering, where our proposed research can make a faster practical impact by targeting software practitioners: a prime example of such venues is, of course, the International Conference on Software Engineering (ICSE).

Materials and supplies: The requested funds are to pay for incidental expenses associated with the research, including presentation materials and publication costs, as well as for manuals and books.

Equipment: Funds are requested in the first year for three Pentium workstations with 1GB RAM and a large and fast hard disk. Initially, these will be used by the graduate students, the PI, and the international collaborator for development and experimentation. Once our algorithms are implemented in the SMART package, they will be made available for remote execution on these workstations, which will be web–accessible. This will allow us to gather essential algorithmic performance data on a wide range of applications.

Participant support costs: Support is requested for our international collaborator, Dr. Gerald Luettgen, to visit the PI twice a year, once for approximately four weeks in the summer and once for approximately two weeks during the winter break. The costs for his participation are anticipated to be $1,250 per week for stipend; $1,850 per trip for air travel, ground transportation, and car rental; and $100 per day for subsistence (hotel and meals).

Indirect costs: The indirect cost rate of the College of William and Mary is 45%, charged on the Modified Total Direct Cost (MDTC), which do not include equipment and tuition.

NOTE: Actual cost figures mentioned above are estimates for the first year; we assume an increase of 5% per year for the second and third years.
Facilities, Equipment and Other Resources

The Department of Computer Science at the College of William and Mary provides a network of fast Pentium workstations running the Linux operating system, plus several Sun workstations for specialized tasks, running various flavor of Unix. These are available to the graduate students to perform both their classwork and their research and are connected through Fast Ethernet. In addition, each faculty member's office has a Pentium workstation.

For advanced work on distributed-memory and shared-memory algorithms and their performance study, the department, as part of the College of William and Mary Computational Science Cluster, was recently awarded a Major Research Instrumentation grant from the National Science Foundation to acquire a Beowulf-style cluster with three subsystems. Two of these subsystems are Pentium-based and have 32 and 64 processors, respectively, as well as different interconnection networks, while the third subsystem contain four Sun multiprocessor machines with four processors each.