(Generalized Decision) Diagrams supersede existing decision diagrams and are closed over $\{\cup, \cap, \setminus, \times\}$

Malcolm Mumme

Abstract

Symbolic model checking tasks, such as state space exploration, bisimulation, lumping, relational composition, and property checking, often employ decision diagrams (DDs) to encode large models, and algorithm libraries, such as TeDDy, to build and analyze them. A variety of DD types have evolved for various situations. *Fully-reduced* DDs are excellent for encoding state-space tuple-sets, while the extreme compactness of *Fully-Identity-reduced* DD (FIDD) encoding is preferred for interleaved transition relations. Other forms, such as *Binary Decision Diagrams* (BDDs), *Extensible Multi-way DDs* (EMDDs), and various kinds of *Edge-Valued Multi-way* DDs (EVMDDs) are each preferred in other special circumstances. Many DD types, such as EMDDs and BDDs have domains closed under set theoretic operations, such as union, intersection, complement, and Cartesian products, while others, such as FIDDs are not, complicating their use. Different DD types encode partially-overlapping domains, impeding translation between different DD types. The type of DD employed must be manually chosen, and in complex model checking situations, compact encoding requires hybrids of the above DD types, often necessitating library extension. A single DD type, with a domain covering extant DD domains while also being closed under set theoretic operations and having the compactness advantages of existing DD types, would be preferred.

(Generalized Decision) Diagrams (GDDs), appear to have such advantages. I recently proved that normalized (canonical) GDDs encode all finite-*Bundle*-wise constant functions, which includes the tuple-set domains of all the above DD types and is closed under set-theoretic operations. Additionally, GDDs appear to have the compactness advantages of each of these encodings for those cases where the previous encoding had an advantage. Due to the heuristic nature of DD methods, however, the actual efficiency of GDDs in practical model checking tasks cannot be guaranteed by theoretical analysis. Compactness advantantages of GDDs are also not guaranteed to outweigh the additional complexity of GDD algorithms in practice. Thus, the use of GDDs as an improvement to practical model checking systems cannot be recommended until their performance and compactness advantages are verified by experimental analysis involving comparison with existing DD types.

I propose to measure performance and compactness differences between the use of GDDs and other DD encodings in various model checking tasks. This includes (1) preparing an algorithm library, TeDDy, supporting GDDs, with a modernized API and efficient use of common multicore processors through library-level parallelism, and (2) applying TeDDy GDDs to several model checking tasks, including saturation-based state-space exploration, my existing fully-symbolic bisimulation algorithm, and a novel bisimulation technique described below.

I additionally propose to extend my existing model checking research in the following three directions: (1) I will explore using 'uncertain' transition relations to augment a transition system so as to effectively improve the overall locality of transition relations, to improve my saturation-based weak bisimulation algorithm, (2) I will study a new fully-symbolic lumping algorithm that uses the saturation heuristic, in a manner similar to our fully-symbolic weak bisimulation algorithm, and (3) I will study a new parallel saturation technique, having additional novel structural recursion that is organized according to both processor locality and model locality.

I hope to experimentally show that GDDs outperform other symbolic encodings in nearly all practical cases. Thus, this research hopes to provide a basis for recommending the use of GDDs in practical model checking.

1 Introduction

DDs compactly encode large sets and relations used in model checking and functions used in stochastic model checking. This research aims to show that *(Generalized Decision) Diagrams* (GDDs) provide a superior replacement for many extant DD types as used in model checking.

I briefly summarize the state of the practice with respect to sequential DD programming and parallel DD programming, describing potential advances with my approach.

1.1 Decision Diagrams

As originally described, a *binary decision diagram* (BDD) is a directed graph or shared binary tree used to encode a function of multiple (K) binary variables (a K-tuple of booleans), where the range is also boolean [6](§7.37). In the worst case, a BDD is a binary decision tree and occupies space proportional to the size of the table of the function encoded. In some practical cases, many subtrees of a decision tree are identical, so the BDD encoding occupies considerably less space. A binary decision tree is a constant-depth binary tree with the depth being the number of variables input to the encoded function, and the leaves being boolean range values. We do not consider *reduced* DDs, which do not have constant depth, until Section 2.1.3. The *level* of a node is the distance from the node to a leaf. Each level of the tree corresponds to one of the input variables, according to the *variable ordering.* Each node of the tree has 2 outgoing edges, labeled 0, and 1, corresponding to the values of the input variable associated with the level of that node.

An encoded function f_A (of type: $\mathbb{B}^K \rightarrow \mathbb{B}$) is evaluated for its tuple X ($\in \mathbb{B}^K$) of K boolean arguments $X_1, \ldots, X_K$ using its decision tree or BDD encoding A as follows:

1. let b refer to the root of A
2. for each $i \in K, \ldots, 1$, in decreasing order

   if X_i is true let b refer to the node from b reached by following the edge labeled 1,

   otherwise let b refer to the node from b reached by following the edge labeled 0.
3. b now refers to a leaf which is the result of the function.

When illustrating decision diagrams, the edge labels are usually indicated in the body of the node from which their corresponding edges proceed.

Fig. 1 shows both encodings of the function $f(X) = (X_3 \oplus X_2) \vee X_1$, illustrating that the BDD encoding is more compact (requires fewer nodes).

These illustrations also provide a hint that decision trees resemble finite automata accepting languages consisting of a set of finite strings of fixed length, and the leaves correspond to terminal nodes and are additionally labeled with range values. Decision diagrams then correspond to minimized automata.

Decision diagrams are thus a form of compressed representation of their encoded functions, and thus may not always be more compact than an explicit representation. To efficiently detect sharable sub-trees, it helps for an encoding to be *canonical.* A canonical encoding is one that has at most a single encoding for a given encoded function, so that there are no two encodings for the same function. With a canonical encoding, no function is ever encoded redundantly. With many simple forms of decision diagram, canonicity is easy to prove. Many complex forms of decision diagram have been proposed, which in many cases are not canonical. GDD encoding was proven canonical [36](§7.48), and so supports efficient sub-tree sharing.

The performance of algorithms on decision diagrams benefits from the structure and compactness of their encoding. New trees/nodes are constructed by the *unique*(...) function, which receives as input an ordered collection of nodes for use as the child nodes of the newly constructed node. If an existing node

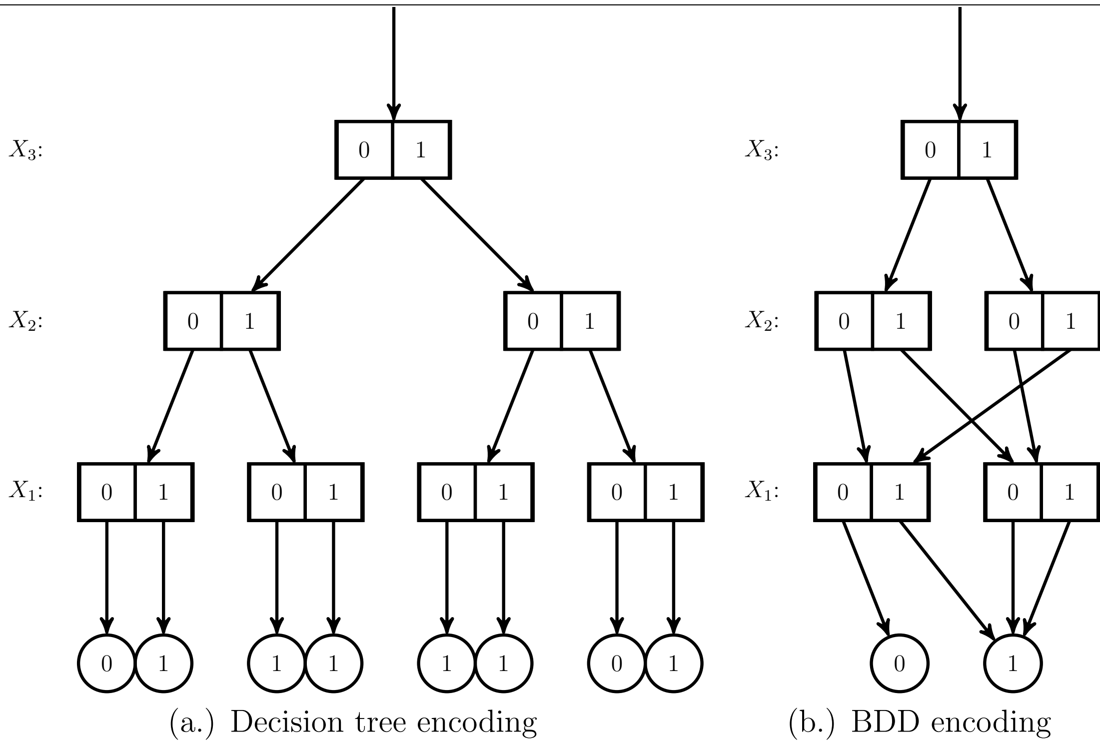


Fig. 1: Encodings of $f(X) = (X_3 \oplus X_2) \vee X_1$

already has the same child nodes in the same order, the existing node is returned instead, invisibly implementing sharing among sub-trees. Fundamental algorithms on decision diagrams are typically recursive tree-style traversals that depend for efficiency on memoization of function calls. For illustration, I will give a BDD algorithm for union of two sets. Sets are represented as encoded characteristic functions, so the characteristic function $f_{a\cup b}$ of the result $a \cup b$ of the union of a and b represented as their characteristic functions f_a and f_b, respectively, is $f_{a\cup b}(x) = f_a(x) \vee f_b(x)$, that is, simply the vector 'or' operation on the individual components of the encodings of f_a and f_b. The BDD union algorithm is simply an adaptation of this definition applied to a tree-like encoding of such an array.

This BDD algorithm for union of two sets (a and b of K-tuples) is given here:

memoized Function $union(a, b, K) : BDD \times BDD \times \mathbb{N} \rightarrow BDD$ is:

- if $(K = 0)$ then: — $a, b \in \mathbb{B}$
  - $union(a, b, K) = a \vee b$
- else $(K > 0)$
  - Variable $children$ is Array[$\mathbb{B}$] of BDD — An array of 2 BDDs
  - Variable $c : BDD$
  - For all $X_K \in \mathbb{B}$ — Loop over possible values of X_K.
    - Let a', b' = child X_K of a, child X_K of b
    - Assign $children[X_K] \leftarrow union(a', b', K-1)$ — recursive subdivision for part X_K
  - Assign $c \leftarrow unique(children)$
  - $union(a, b, K) = c$

Complement of sets represented by their encoded characteristic functions uses the formula $f_{\overline{s}}(X) = \neg f_s(X)$, and can be easily implemented in BDD libraries as an operation which (modulo canonicity) copies a BDD, and when reaching the leaves, substitutes '1' for '0', and '0' for '1'.

Thus BDD-encoded sets are closed under union and complement, and obviously all other set operations, such as conjunction, disjoint union etc., as well as cartesian products.

Many additional refinements are employed in practical versions of the above algorithm.

It has been found that in encoding transition relations of asynchronous state transition systems, it is preferable to interleave the variables of the 'from' state with variables of the 'to' state, often resulting in considerable space savings for the encoding. Usually, such encodings are additionally greatly simplified by the use of FI-reduced diagrams (FIDDs) (Section 2.1.5), so much so that use of FIDD encoding is a practical necessity in many model checking problems. Unlike many other types of decision diagram, FIDD encodings are not closed over union and complement, and additionally do not have strong theoretical support in the literature. Nevertheless, model checking frequently requires operations to combine FIDD encodings with each other and with other DD encodings, causing unnecessary complexity and confusion for algorithm libraries attempting to support such encodings, and for users of such libraries.

This proposal introduces GDDs, which unify both the compactness advantages and the theoretical advantages of all these DD types. In particular, GDDs are closed over the set operations, yet also have the space saving advantages of FIDDs and encode all sets encoded by FIDDs and the other DD types we discuss. This discovery and the recent proof of the theoretical properties of GDDs enables the current proposal for comparing the practical advantages of GDDs with those of prior DD encodings.

Parallel processing has been employed with some success for the processing of DDs used in model checking. Parallel processing has allowed model checking of larger models than possible with sequential processing, but has not produced wished-for massive scalable parallel speed-ups for model checking. By efficiently using multicore processors, TeDDy supports future efforts in attaining usable parallel speed-ups.

1.2 Structure and overview of this Proposal

In Section 2 (Background), Section 2.1 includes greater detail about some of the great variety of decision diagram types that have been studied since the origin of BDDs, as well as discussion of algorithms and applications of decision diagrams relevant to this proposal, while Section 2.2 discusses the evolution of library metaprogramming, Sections 2.3-2.4 briefly explain saturation, bisimulation, and lumping, which are model checking techniques I have studied and propose to continue exploring as a basis for comparing GDDs with other DD types, Sections 2.5-2.6 discuss use of parallelism in DD algorithm libraries and in model checking algorithms, and Section 2.7 briefly discusses the space overhead associated with use of DD encodings. Section 3 (ProposedContributions) explains the proposed work in detail, specifically: In Section 3.1, GDDs, my novel form of decision diagram, are defined, their semantics and consistency rules are explained in Sections 3.1.1-3.1.2, canonicity for GDDs is formally defined in Section 3.1.3, which also references the proof of canonicity, closure properties are discussed in Section 3.1.4, which also references the relevant proof that GDDs encodings cover the previously discussed encodings, a few apparently simple improvements are discussed in Sections 3.1.5-3.1.7 which have the cumulative effect of providing to GDDs the compactness advantages of all previously discussed encodings, and the GDD *Union* algorithm is referenced in Section 3.1.8. In Section 3.2 my proposal for DD library API modernization is illustrated, and Section 3.3 proposes to use GDD algorithms in model checking techniques such as bisimulation and saturation. Section 4 potentially proposes additional explorations contingent on available time, while Section 5 (Future work) briefly outlines obvious future extensions of the currently proposed work. Section 6 (Schedule) provides a detailed schedule.

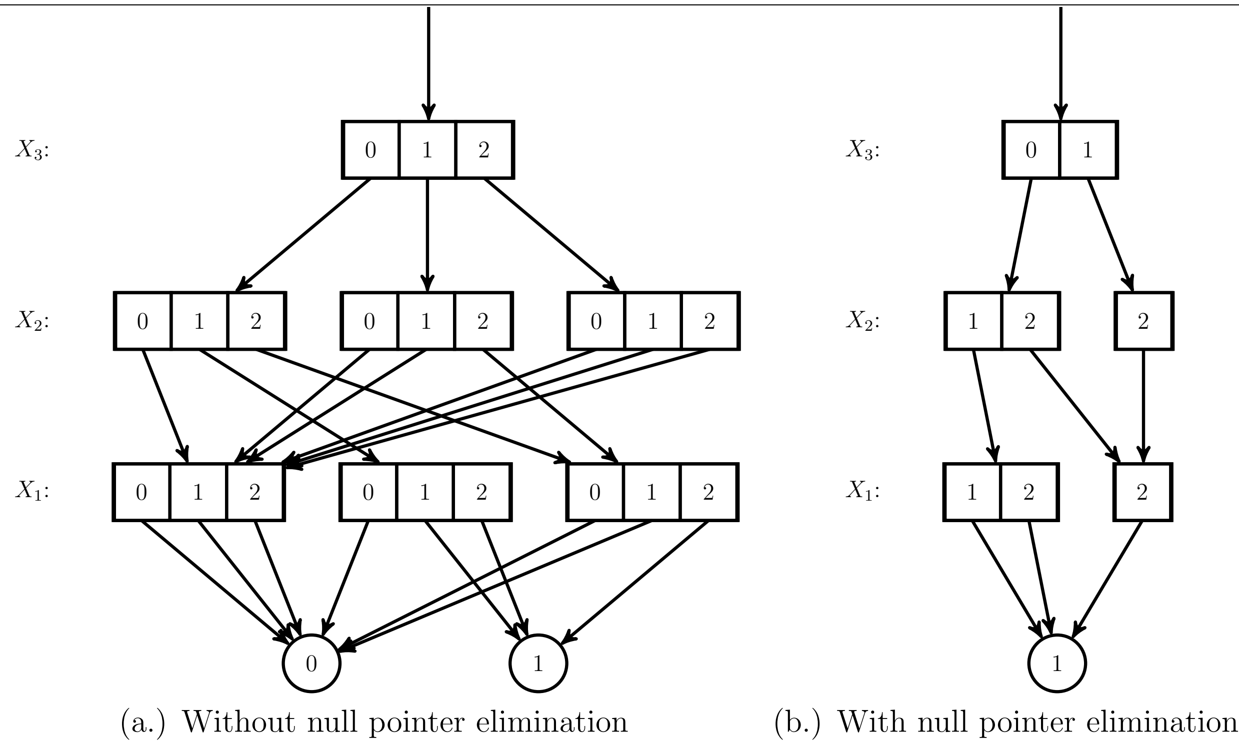


Fig. 2: MDDs encoding the function $f(X) = (X_3 < X_2 \leq X_1)$ given $X_1, X_2, X_3 \in \{0, \ldots, 2\}$

2 Background

2.1 Multi-Way Decision Diagrams

Multi-way decision diagrams (MDDs) generalise BDDs by expansion of the types of the members of the tuple domain, so that the domain is $D_K \times \ldots \times D_1$, where, for each $k \in 1, \ldots, K$, D_k is any chosen finite set, assumed to be $\{0, \ldots, Max_k\}$ (where each $Max_k \in \mathbb{N}$), without loss of generality, for a suitable selection of $Max_K, \ldots, Max_1$.

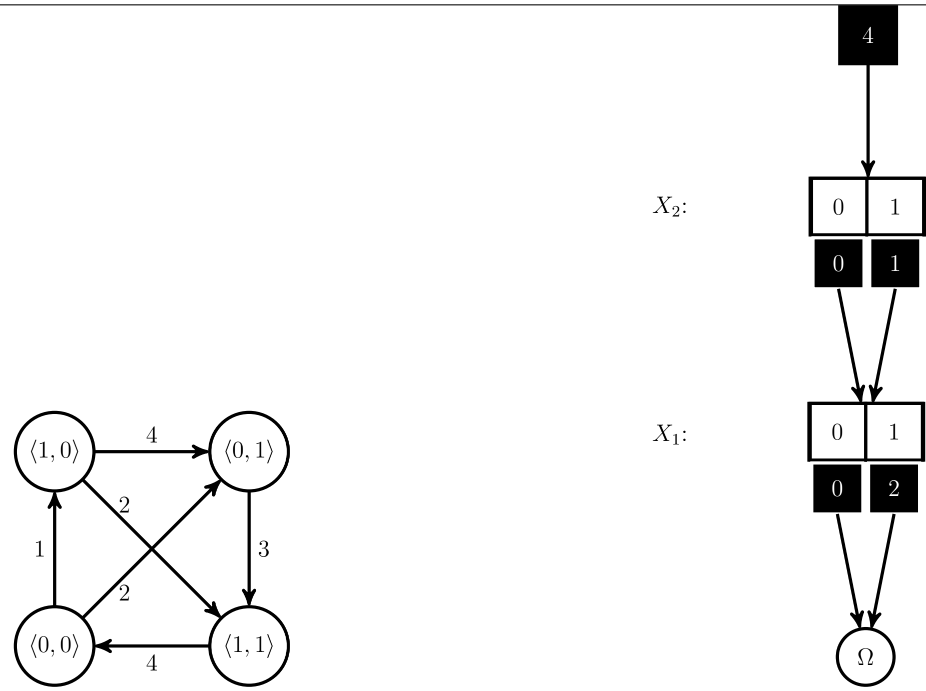
Thus MDDs encode all functions of type $\{0, \ldots, Max_K\} \times \ldots \times \{0, \ldots, Max_1\} \to \mathbb{B}$.

Like BDDs, MDDs are directed graphs resulting from sharing nodes in a tree, where the tree is similar to an automata for accepting a language of strings of fixed length. An MDD is identical to a BDD when $Max_k = 1$ for all $k \in K \ldots 1$. When $Max_k > 1$ for some k, nodes at level k have $Max_k + 1$ outgoing edges, each labeled with a distinct member of $\{0, \ldots, Max_k\}$. As with BDDs, drawings of MDDs show edge labels in the body of the node from which the edge proceeds. Inasmuch as tuples of any finite type can be encoded as boolean tuples, it is of course possible to encode functions of type $\{0, \ldots, Max_K\} \times \ldots \times \{0, \ldots, Max_1\} \to \mathbb{B}$ as BDDs instead of MDDs, however there are many cases where it is less natural to do so, and where the MDD encoding of such functions is more efficient to use.

Fig. 2.a shows an MDD encoding the function $f(X) = (X_3 < X_2 \leq X_1)$ where $X_1, X_2, X_3 \in \{0, \ldots, 2\}$. A commonly used improvement, called *null pointer elimination* to MDDs removes all edges that always lead to the '0' leaf, as well as nodes having only such edges. Fig. 2.b illustrates the memory savings resulting from null pointer elimination applied to an MDD for the same function.

Actual high-performance MDD algorithm libraries employ many additional encoding devices to further reduce memory usage.

Here I give an algorithm for evaluation of an MDD-encoded function (with null pointer elimination):



(a.) Directed graph with weighted edges (b.) EV+MDD encoding of distance from $\langle 0, 0 \rangle$

Fig. 3: EV+MDD encodes distance in directed graph

An encoded function f_A (of type: $\{0, \ldots, Max_K\} \times \ldots \times \{0, \ldots, Max_1\} \rightarrow \mathbb{B}$) is evaluated for its tuple X ($\in \mathbb{N}^K$) of K arguments $X_1, \ldots, X_K$ using its MDD encoding A as follows:

1. let b refer to the root of A
2. for each $i \in K, \ldots, 1$, in decreasing order

   if b has an outgoing edge labeled with the value of X_i, let b refer to the node from b reached by following the edge labeled with the value of X_i.

   otherwise the value of the function is $false$, hence terminate early.
3. b now refers to a leaf which is the result of the function.

2.1.1 Edge-valued Multi-Way Decision Diagrams

The possible range of encoded functions may be extended beyond the booleans $\mathbb{B}$ to a larger range R by simply employing additional values in R for leaf nodes. Such MDDs are called *Multi-Terminal Multi-way Decision Diagrams*, or MTMDDs. MTMDDs are most efficient in certain cases where the range is relatively small compared to the domain. In many other cases, *Edge-Valued Multi-way Decision Diagrams* (EVMDDs) are more compact by many orders of magnitude, and allow the use of ranges that are not explicitly known a-priori.

Each edge, including the root incoming edge, of an EVMDD stores an operation, encoded as an *edge value*. To evaluate the encoded function, an 'accumulator' variable is initialized with a default value, and modified by each operator encountered during traversal of the EVMDD, leaving the function result in the 'accumulator' variable. No specific value is associated with leaves, so only one leaf exists, conventionally labeled Ω.

An EV+MDD is an EVMDD where the default value is 0, and the operation stored at each edge is the addition of the edge value which is in $\mathbb{N}$, while an EV*MDD has a default value of 1, and the operation is multiplication by the edge value. EV*MDDs have been used to compactly encode transition probabilities in Markov process graphs [52](§7.56). EV+MDDs have been used to compactly encode distance information for transition graphs [12](§7.50). Fig. 3 shows (a.) a transition graph, and (b.) an EV+MDD encoding of the distance (from a specific node) for all the nodes in the graph. Note that the edge value for each edge is customarily displayed in a black box at the top of the corresponding drawn edge. Canonicity of EVMDDs is more complex than the case for MDDs, and will not be explained here.

To clarify, the algorithm for evaluation of EV+MDD-encoded functions is given here:

An encoded function f_A (of type: $\{0,\ldots,Max_K\} \times \ldots \times \{0,\ldots,Max_1\} \to \mathbb{N}\cup\{\infty\}$) is evaluated for its tuple X ($\in \mathbb{N}^K$) of K arguments $X_1,\ldots,X_K$ using its EV+MDD encoding A as follows:

1. let b refer to the edge leading to the root of A
2. let v be the edge value of the edge b
3. initialize $r \leftarrow v$
4. for each $i \in K,\ldots,1$, in decreasing order

   if the target of b has an outgoing edge labeled with the value of X_i, let b refer to the edge labeled with the value of X_i.

   otherwise the value of the function is ∞, hence terminate early.

   let v be the edge value of the edge b

   calculate $r \leftarrow r + v$
5. Now r is the result of the function.

2.1.2 Extensible Multi-Way Decision Diagrams

Extensible mult-way decision diagrams (EMDDs) provide encoded functions with limited access to infinite domains. MDDs with K levels encode certain functions of type: $\mathbb{N}^K \to R$ (for some range R).

The encoding is different from plain MDD encoding in that an EMDD node may have any finite number of outgoing edges, and they are (uniquely) labeled with naturals, except there is always one outgoing edge labeled as '*'. The algorithm for evaluating an encoded function f_A (of type: $\mathbb{N}^K \to R$) on its argument X ($\subseteq \mathbb{N}^K$) using its EMDD encoding A is altered to the following:

1. let b refer to the root of A
2. for each $i \in K,\ldots,1$, in decreasing order

   let b refer to the node reached from an edge labeled with the value of X_K (if such an edge from b exists)

   let b refer to the node reached from the edge labeled '*' (otherwise)
3. b now refers to a leaf which is the result of the function.

The specific EMDD encodings supported by extant algorithm libraries may impose some additional restrictions on the collection of labels of edges going out from a node. TeDDy requires that the natural labels be in a contiguous range starting from 0, and, to enforce canonicity, the edge having the largest natural label must not lead to the same node as the edge labeled '*'.

Note that without the contiguity restriction, enforcing canonicity would require that from a given node, the edge labeled '*' must not lead to the same node as any other edge from the given node.

The '*unique*' function is more complex than in the BDD case, as it must discard certain redundant children to enforce canonicity.

The EMDD encoding allows some infinite sets (along with all finite sets) of natural tuples to be represented via their encoded characteristic functions. The complement of such sets may be taken using the same algorithm as for BDD-encoded sets, however the union algorithm for EMDD-encoded sets is more complex, and is given here:

memoized Function $union(a, b, K) : EMDD \times EMDD \times \mathbb{N} \to EMDD$ is:

- if $(K = 0)$ then: (leaf case where $a, b \in R$)
  - $union(a, b, K) = a \vee b$
- else $(K > 0)$
  - Let s be the set of all labels of edges from a or from b
  - Variable $children$ is Map[s] to $EMDD$ (A mapping of edge labels to EMDDs)
  - Variable $c : EMDD$
  - For all $X_K \in s$ (Loop over possible values of X_K (or '*'))
    - Let a', b' = child X_K of a, child X_K of b (use child '*' when child X_K does not exist)
    - Assign $children[X_K] \leftarrow union(a', b', K - 1)$ recursive subdivision for part X_K
  - Assign $c \leftarrow unique(children)$
  - $union(a, b, K) = c$

Of course, practical versions of the above algorithm employ a great many additional refinements.

EMDDs (and EVMDDs, MDDs, and BDDs) can all be used in combination with the following reductions, which, in many important cases, can produce many orders of magnitude of additional improvement in storage space and computation time. Until now, all the types of DD we have seen have edges only between adjacent levels. Thus the level of a node can easily be determined by subtracting 1 from the level of its predecessor. Reductions assign meaning to diagrams where an edge may *skip* one or more levels, by leading from a node at some level k, to a node at some level $j < k - 1$. In these cases, an explicit level number may be stored with each node, so that its level number may be easily determined in the presence of edges that skip levels. Within algorithms, a $level()$ function mapping nodes to $\mathbb{N}$ is employed to give the level of any node.

2.1.3 Fully-Reduced Multi-Way Decision Diagrams

Fully-Reduced MDDs utilize implicit knowledge of the domain when assigning meaning to diagrams where level(s) are skipped. An edge e from a node A at level $k + 1$ which skips level k and leads to a node C at some level $j < k$ is a shorthand for the case where the edge e instead leads to a node B at level k, and where node B has $|D_k|$ children, each uniquely labeled with a member of D_k, and each leading to C (When $D_k = \mathbb{N}$, the shorthand is slightly different). Hence, the output of the encoded function is independent of X_k. Thus, Fully-Reduced MDDs provide a compact way to encode a function that, on

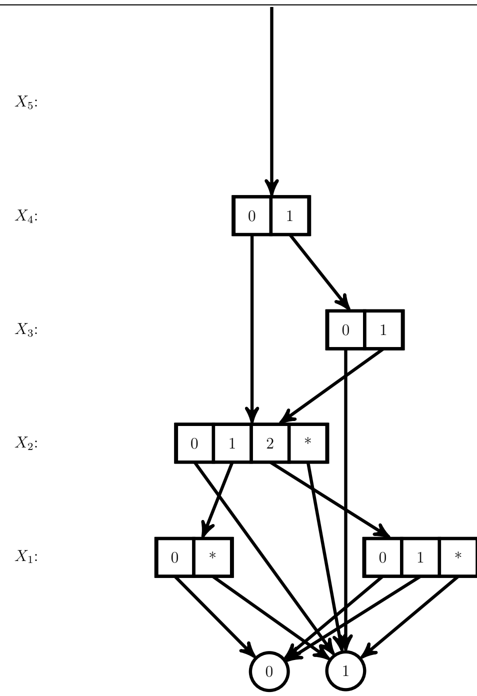


Fig. 4: Fully-Reduced EMDD encoding $f(X) = (X_4 \wedge \neg X_3) \vee (X_2 \leq X_1) \vee (X_2 > 2)$ over $\mathbb{N} \times \mathbb{B} \times \mathbb{B} \times \mathbb{N} \times \mathbb{N}$

some path, ignores a member X_k of its input tuple X, leading to considerable reduction of space when functions frequently ignore many members of their input tuples. Fig. 4 shows a Fully-Reduced MDD with null-pointer elimination that encodes $f(X) = (X_4 \wedge \neg X_3) \vee (X_2 \leq X_1) \vee (X_2 > 2)$ over $\mathbb{N} \times \mathbb{B} \times \mathbb{B} \times \mathbb{N} \times \mathbb{N}$.

The algorithm for evaluating an encoded function f_A on its argument X using its Fully-Reduced MDD (or Fully-Reduced EMDD) (and allowing null-pointer elimination) encoding A is additionally altered as follows:

1. let b refer to the root of A (or the unique leaf of A, if all levels are skipped)

2. for each $i \in K, \ldots, 1$, in decreasing order

   if $i = level(b)$

   let b refer to the node reached from an edge labeled with the value of X_i (if such an edge from b exists)

   otherwise:

   let b refer to the node reached from the edge labeled '*' (if such an edge from b exists)

   otherwise the value of the function is $false$, hence terminate early.

3. b now refers to a leaf which is the result of the function.

Note that this algorithm is an extension of the corresponding algorithms for both EMDDs and for MDDs with null-pointer elimination. As the complexity of the decision diagram types increases, the conditions for the canonicity of a decision diagram become more complex. For the sake of brevity I will not discuss the canonicity conditions for diagrams with reductions.

2.1.4 Identity-Reduced Multi-Way Decision Diagrams

Identity-Reduced diagrams provide space reduction when encoding a function that sometimes depends on whether or not some tuple member X_k is equal to the preceding tuple member X_{k+1}. In those times, such a function returns false when $X_k \neq X_{k+1}$. In this case, an edge that skips from node A at level $k+1$ to some node C at level $j < k$ indicates, for tuples that invoke this path, that the value of the function is $false$ if $X_{k+1} \neq X_k$. Identity reduction is generally not used alone, but in combination with full reduction as discussed next.

2.1.5 Fully-Identity-Reduced Multi-Way Decision Diagrams

Fully-Identity-Reduced MDDs incorporate 'full' reduction at even levels and 'identity' reduction at odd levels. A 2-level Fully-Identity-Reduced MDD, where both levels are skipped, and the edge leads to the *true* node, compactly encodes the identity relation between X_2 and X_1, independently of the (necessarily common) domain of X_2 and X_1. This so-called identity pattern may be used to advantage in multiple parts of a Fully-Identity-Reduced MDD. Doing so has been found to greatly reduce space requirements for interleaved encodings of transition relations where the *support* of the transition relation omits many variables. The support of a transition relation is the set of variables used in or changed by the transition relation. Thus it has found application in practical model checking systems. These advantages must not be ignored when considering encodings for such transition relations, and my novel structures retain these advantages in certain variants.

The algorithm for evaluating an encoded function f_A on its argument X using its Fully-Identity-Reduced MDD (or Fully-Identity-Reduced EMDD) (and allowing null-pointer elimination) encoding A is as follows:

1. let b refer to the root of A (or the unique leaf of A, if all levels are skipped)
2. for each even $i \in \{K, K-2, \ldots, 2\}$, in decreasing order

    (process an even (Fully-Reduced) level):

    if $i = level(b)$ (the Fully-Reduced level is not skipped)

    let b refer to the node reached from an edge labeled with the value of X_i (if such an edge from b exists)

    otherwise:

    let b refer to the node reached from the edge labeled '*' (if such an edge from b exists)

    otherwise the value of the function is $false$, hence terminate early.

    (process the odd (Identity-Reduced) level just below):

    if $i-1 = level(b)$ (the Identity-Reduced level is not skipped)

    let b refer to the node reached from an edge labeled with the value of X_{i-1} (if such an edge from b exists)

    otherwise:

    let b refer to the node reached from the edge labeled '*' (if such an edge from b exists)

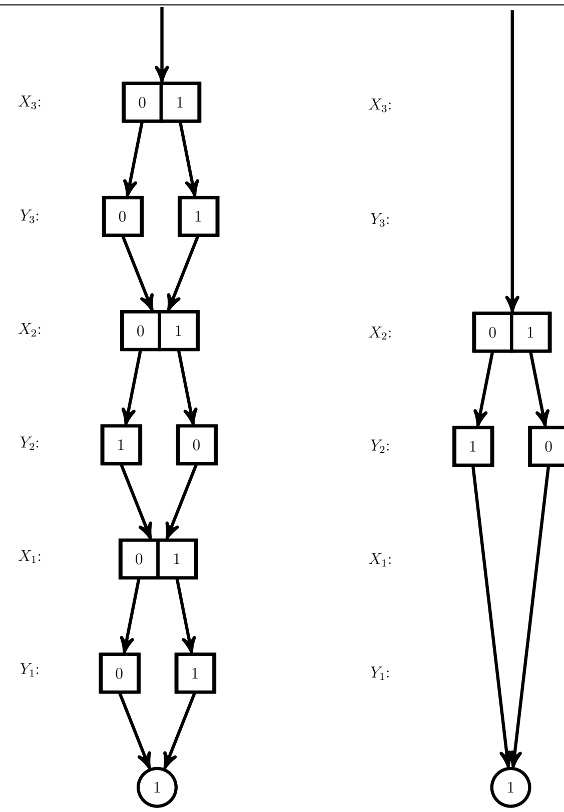
    otherwise the value of the function is $false$, hence terminate early

    otherwise (the Identity-Reduced level is skipped)

    if $X_i \neq X_{i-1}$, then the value of the function is $false$, hence terminate early

3. b now refers to a leaf which is the result of the function.

Fig. 5 illustrates the improvement gained using Fully-Identity-Reduction for encoding interleaved relations having a support that is small compared to the total number of variables.



(a.) Fully-Reduced (b.) Fully-Identity-Reduced
(Both with null-pointer elimination)

Fig. 5: Fully-Identity-Reduced MDDs encoding $f(X) = (\langle Y_3, Y_2, Y_1 \rangle = \langle X_3, \neg X_2, X_1 \rangle)$ over $\mathbb{B}^{2^3}$. Note that all node levels in the Fully-Identity-Reduced encoding are in the support $(\{X_2, Y_2\})$ of f

2.1.6 GDD Multi-Way Decision Diagrams

This novel data structure encodes all sets and functions encoded by the above mentioned styles of DD, providing a unified framework in which to exploit all their advantages, and is the subject of Section 3.1.

2.2 C++ template metaprogramming and library interface improvement

Extant libraries that support the above-mentioned kinds of decision diagram are often somewhat cumbersome to use for building model checkers, as compared with what one might hope for, given that usually one merely needs to efficiently provide certain tuple-set-based operations.

Of special concern is the fact that, for the sake of efficiency, tuple-set operations must sometimes be performed where different arguments are different kinds of DD. Until now, these situations have been handled individually by manually defining a new function to correctly calculate the desired function given the specific kinds of DDs used as input and desired as output. Thus, these libraries must be incrementally extended by their users when new combinations of parameter types are necessary. This situation results in a great may manually constructed functions, each of which must be tediously debugged, etc., as well as in libraries where the API is bloated with many functions with similar purposes. This situation motivates our novel invention of GDDs, which have the theoretical and practical advantages of all of the above kinds of DD.

I propose to remedy the particular problem of having too many manually defined functions, through the use of GDDs and parametric variants thereof to describe the types of DDs and instantiate all associated operations through template metaprogramming from a relatively small base of manually generated code.

I further propose to improve the API of GDD libraries through the use of C++ template metaprogramming in a way that allows executable definitions to be modeled more closely on the higher-level pseudocode on which they are based.

Procedural *metaprogramming*, or the generation of code at compile time by user code, has been used in procedural languages ever since *macro* was added to the Lisp programming language, for the purpose of generating efficient code for specialized 'mini-language' notations embedded within procedural language code [34](§7.38). Later procedural languages, such as PL/1 and C also adopted the use of macros for code generation, although those macro implementations were relatively text-oriented and could easily generate unintended multiple definitions for a given name. The relative messiness of macros prompted the invention of 'hygienic' macro-expansion for languages in the Lisp family [30](§7.39)[2](§7.40)[5](§7.41) , while Ada and Java allow only *generics*, which have considerably less flexibility than macros, but are relatively safe [29](§7.44)[23](§7.45). The C++ programming language uses templates and type-based specialization to obtain many more of the benefits of macros in a relatively safe way [47](§7.42). The Scala programming language was created with extended support for metaprogramming in the form of implicit parameters, manifests, language feature virtualization, syntactic flexibility, and other features [39](§7.21)[35](§7.22). Scala would be the preferred language for this research if not for concerns about efficiency, as Scala apparently compiles to Java bytecode. It has however been shown that C++ template specialization can be used to perform almost arbitrarily complex computation at compile time [44](§7.47). Thus C++ is in principle capable of supporting the kind of notational improvements I would like to include although with less syntactic convenience than what is possible with Lisp or Scala. The recent availability of compilers for C++11 further simplifies the task of metaprogramming in C++ [48](§7.43)[45](§7.46). Hence, I believe a C++11 GDD library interface using template metaprogramming, would fit well the needs of the proposed model-checking research.

2.3 Saturation

The *Saturation* heuristic applies to fixed-point iteration problems where the solution is closed over many (preferably simple) operators. These problems may be solved by repetitively applying the operators as steps in any order, where applying any step does not diminish the progress due to later application of another step. That is, some steps applied in a certain order will cause the problem to be solved, and extra 'wrong' steps that occur in the process do not prevent solution.

The Saturation heuristic is controlled by a given ordering of the possible steps, usually corresponding to the approximate cost of a step. The heuristic simply applies inexpensive steps until they yield no change in the problem state, after which a more expensive step is tried. That is, inexpensive steps have priority over expensive steps. So, a step is applied only if application of all less expensive steps yields no change in the problem state. Many useful refinements to Saturation in the symbolic context are described in [10](§7.1).

Saturation has been found to be very effective for exploration of (certain kinds of) finite state spaces, when the state spaces are encoded symbolically (using DDs). Here, the problem state is encoded symbolically as a set of discovered states, and the steps are applications of state transition relations, also encoded symbolically. The SmArT tool is known for its use of Saturation to rapidly explore Petri Net state spaces with over 10^{20} states within seconds [11](§7.51).

Saturation has also been found effective for other model checking-related tasks, such as:

1. Calculating distance in large transition graphs [12](§7.50)

2. Strongly Connected Components via Transitive Closure of transition relations in large transition systems [55](§7.53)

3. Bisimulation with many (over 10^9) equivalence classes [37](§7.2)[38](§7.3)

2.4 Fully-symbolic algorithms for bisimulation and lumping

The author has previously explored application of the saturation heuristic to symbolic bisimulation, with some good results [37](§7.2)[38](§7.3). Significant progress in application of saturation to symbolic lumping requires using the advantages of Fully-Identity-Reduced MDDs in combination with Edge-Valued MDDs to model probabilistic transition relations, and partially motivates the current proposal. These efforts are briefly described below.

2.4.1 Bisimulation with deterministic transitions

A *bisimulation*[40] relation relates extensionally equivalent states in a labeled transition system. Exact extensional equivalence between states in given by the maximum bisimulation relation. The maximum bisimulation relation[33], $\sim\ \subseteq S \times S$ between sets of states S of a transition system with transition relations E (where $\forall(\overset{\alpha}{\to}) \in E : (\overset{\alpha}{\to}) \subseteq S \times S$), is defined as the largest equivalence relation B on S satisfying:

$$\forall(\overset{\alpha}{\to}) \in E : \forall\langle p, q\rangle \in B : \forall p' \in S : p\overset{\alpha}{\to}p' \Rightarrow \exists q' \in S : q\overset{\alpha}{\to}q' \wedge \langle p', q'\rangle \in B$$

In late 2008, I observed that when all transition relations were deterministic, as with individual transitions in Petri Nets, we have: $\forall(\overset{\alpha}{\to}) \in E : \forall\langle p, q\rangle \in \overline{\sim} : \langle(\overset{\alpha}{\to})^{-1}(p), (\overset{\alpha}{\to})^{-1}(q)\rangle \in \overline{\sim}$, thus, $\overline{\sim}$ is closed over $(\overset{\alpha}{\to})^{-1} \times (\overset{\alpha}{\to})^{-1}$ for all $(\overset{\alpha}{\to}) \in E$.

I also showed that $\overline{\sim}$ could be calculated by initializing a variable $\overline{B}$ to all pairs in $S \times S$ with different enablements or colorings, then applying the transitive closure of $(\overset{\alpha}{\to})^{-1} \times (\overset{\alpha}{\to})^{-1}$ to $\overline{B}$, iterating over all $(\overset{\alpha}{\to}) \in E$ according to the saturation heuristic.

The resulting 2011 paper [37](§7.2), showed good results for using the saturation heuristic to implement the transitive closure in the above strategy, resulting in the fastest known bisimulation algorithm when there are many resulting equivalence classes and the transition relations are deterministic.

2.4.2 Weak bisimulation

Weak bisimulation allows the possibility of 'invisible' transitions, which may occur without removing an input symbol. The largest weak bisimulation may be calculated using ordinary bisimulation algorithms, provided that the transition relations are preprocessed by appending the transitive closure of all invisible transition relations to each visible transition relation.

This preprocessing typically results in a transition system with nondeterministic transition relations. Nondeterministic transition relations may also arise with Petri Nets if multiple Petri Net transitions are given the same label in the transition system. Thus it is quite desirable to be able to efficiently calculate the largest bisimulation relation ($\sim\ \subseteq S \times S$) when some transitions are nondeterministic.

Closer analysis in 2009 showed that the definition of bisimulation can be reformulated as:

$$\begin{gathered}(\sim) \text{ is the largest relation } B \subseteq S \times S \text{ on } S \text{ satisfying:}\\ \forall(\overset{\alpha}{\to}) \in E : B = B \setminus \{\langle p, q\rangle | (\exists p' : (p\overset{\alpha}{\to}p' \wedge \neg\exists q' : q\overset{\alpha}{\to}q' \wedge p'Bq')) \vee (\exists q' : (q\overset{\alpha}{\to}q' \wedge \neg\exists p' : p\overset{\alpha}{\to}p' \wedge q'Bp'))\}\end{gathered}$$

and that this formula was actually compatible with Saturation based solution. My 2013 paper [38](§7.3), showed good results for using the resulting saturation-based algorithm on a variety of bisimulation problems. Important weaknesses were also identified, such as inefficiency in cases where some transition relations have very large support, as is often the case with weak bisimulation.

2.4.3 Lumping

Lumping is analogous to bisimulation, except that it calculates extensional equivalence between states of Markov systems rather than states of labeled transition systems. Lumping uses probabilistic transition matrices Q of type $S \times S \to \mathbb{R}$, instead of relations of type $S \times S \to \mathbb{B}$. The lumping equivalence relation may be defined as the largest equivalence relation B satisfying:

$$\forall Q \in E : \forall \langle p, q \rangle \in B : \forall r \in S : \sum_{p' | p' B r} Q(p, p') = \sum_{q' | q' B r} Q(q, q')$$

There is some similarity between symbolic lumping algorithms and symbolic bisimulation algorithms [54](§7.54)[16](§7.49). We expect that the enhanced TeDDy library will provide the necessary flexibility for attempting to solve the symbolic lumping problem, through adaption of our symbolic bisimulation algorithms.

2.5 Parallel DD algorithms

Model checking is frequently a computationally demanding task, so that space and speed improvements provided by symbolic methods do not suffice for all problems of interest. Parallel processing is another avenue by which researchers attempt to expand the range of solvable problems. A variety of approaches, for incorporating parallelism into symbolic model checking, have been explored. Most involve either of the following approaches:

1. Algorithm-level parallelism, where a (possibly sequential) DD library is used on each processor by an algorithm that employs some parallelism techniques to utilize multiple processors.

2. Library-level parallelism, where a (possibly sequential) algorithm invokes a parallelized DD library so that requested DD operations will be executed by the library code using multiple processors.

The RooMY[31](§7.9)[32](§7.10) system illustrates an interesting hybrid of both techniques. Kunkle et al implemented the RooMY system, which effectively utilizes the disks of multiple processors to carry out very large computations. They appear to use some parallelism-like techniques for organizing computation in a way that makes it less sensitive to the latency of lengthy disk accesses. By coherently grouping disk accesses, and using large RAM buffers, the efficiency of disk access is greatly improved, especially when compared to the use of disk as virtual memory for RAM-oriented algorithms. Kunkle et al then implemented a BDD library on top of RooMY, and subsequently solved some problems that could not previously be solved with RAM-based DD algorithm libraries.

This is typical of the existing results for model checking using parallel DD libraries. In most cases, parallel implementations of a parallel algorithm obtain modest speedup compared with sequential implementations of the same algorithm, while comparison with carefully optimized sequential algorithms shows no significant speedup. Also, in many cases, a parallel algorithm is able utilize the RAM (or disks) of multiple machines to solve problems not solvable on a single processor.

There are roughly two strategies used to enable library level parallelism in parallel DD libraries, as described in the following two sub-subsections.

2.5.1 Distribution by level

Even in reduced decision diagrams, typically, the vast majority of edges lead from a node at one level to a node at the next lower level. As algorithms on DDs typically involve traversing the DD graphs through their edges, there is considerable temporal locality associated with connected nodes. In a distributed-memory processor network, distribution by level attempts to exploit this property by locating connected nodes on 'nearby' processors, meaning either the same processor, or processors sharing a physical direct communication link or a shared memory. [46](§7.15) uses this technique. In some cases, this associates a separate level of DD nodes with each processor. Although the matching of locality between the DD nodes with the locality of communication in the processor network produces a desirable limit on communication costs, the hoped-for speedups tend to not materialize. The algorithms for DD operations are usually organized as depth-first or breadth-first traversals of the DD graphs. In the depth-first case, the algorithm is intrinsically serial, so that no speedup is gained from using multiple processors. In the breadth-first case, parallel traversal activities may be spawned for each child of a given node, however this distribution scheme tends to place all of those child nodes in the same processor, again limiting opportunities for parallelism.

2.5.2 Distribution by function of state

These schemes place DD nodes among processors according to some function of their contents. In the case of library-level parallelism the function may be a hash involving the addresses of the node's children, so that the quasi-random node distribution maximizes the chances for parallel operations on any given collection of nodes. As most operations involve many nodes, this scheme is likely to produce much demand on a distributed communication network. In algorithm-level parallelism, different parts of a (partitioned) set may be stored as separate DDs and each DD placed by an algorithm-aware scheduler. This scheme may help to reduce inter-processor communication, but may encode a set less compactly due to the partitioning into multiple DDs.

2.6 Parallel symbolic state space exploration

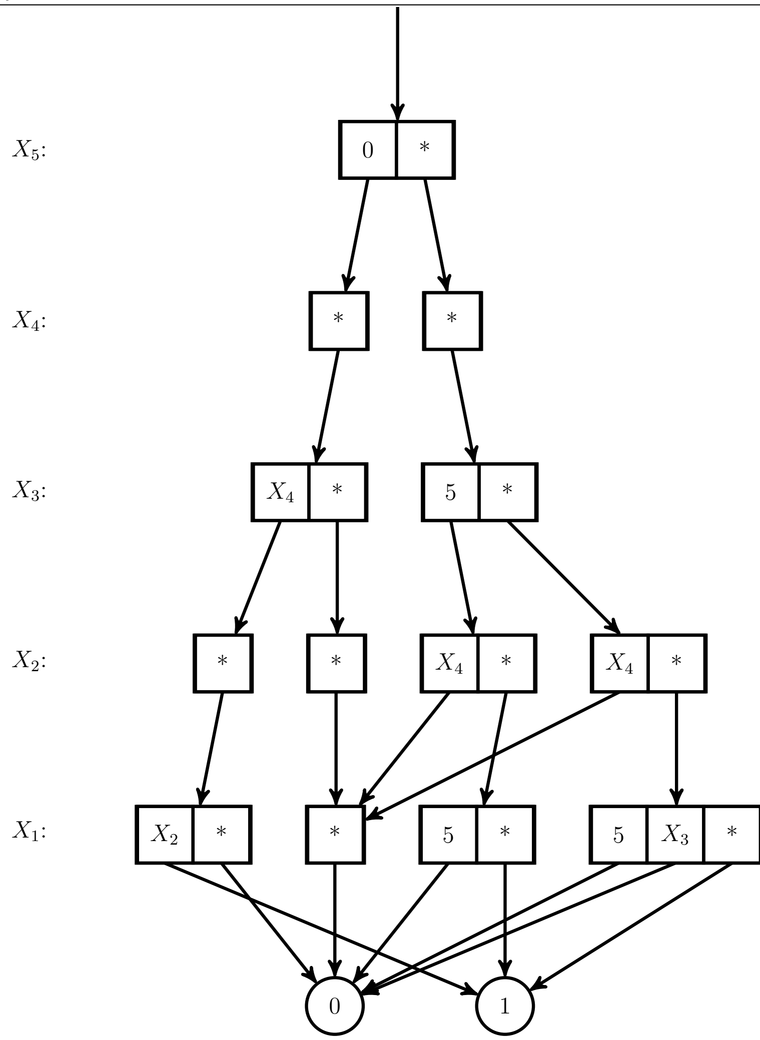
Saturation provides the fastest exploration of finite states spaces currently available on sequential machines. Because Saturation is so efficient, uses recursive procedures and strenuously avoids unnecessary work, parallelization of Saturation has met with limited success. Ideas used to parallelize saturation have resulted in improvements to sequential saturation, nullifying potential parallel speed-ups [17](§7.13)[13](§7.14)[19](§7.17)[21](§7.18)[18](§7.19). So far, parallel saturation has provided access to more memory than what is available to individual computers, enabling solution of larger problems, but not providing hoped-for scalable parallel speed-up [13](§7.14)[19](§7.17)[21](§7.18)[18](§7.19).

Efforts by Grumberg et al, at Technion have obtained some useful speed-up using an approach similar to distribution by function of state described above [28](§7.4)[26](§7.5)[3](§7.6)[25](§7.7)[22](§7.8).

2.7 Space overhead of decision diagrams

Although the use of DDs to encode sets often saves space and time by many orders of magnitude, compared to an element list encoding, it is a heuristic compression technique, and space saving is not guaranteed. In the worst case, only minimal sharing occurs, so that a DD encoding of a set occupies more space than the corresponding element list encoding. This is especially true in the case of very small tuple-sets of large tuples.

As a simple example, consider the case of encoding the set $\{\langle 1,0,1,1,0,1\rangle\}$, as a BDD, and as an element list. The set has only one element, so the element list has a single node, of unspecified complexity, although 6 bits should suffice to encode the element in this case. The (un-reduced) BDD encoding, however, will have 6 levels (hence 6+ nodes), each of which must store 2 pointers. Thus we can see that un-reduced DDs have a space overhead proportional to the number variables in each tuple, for encoding tuple-sets. This overhead is not onerous if one is using DDs to encode sets with very many element tuples each of reasonable size. This overhead is excessive when encoding small sets of large tuples, and may account for the general lack of interest in symbolic encodings outside the model checking and logic programming community.



(a.) Quasi-reduced GDD encoding (b.) Fully-reduced GDD encoding

Fig. 6: $f(X) : \mathbb{N}^5 \rightarrow \mathbb{B} = (X_5 = 0 \wedge (X_4 = X_3 \wedge X_2 = X_1)) \vee (X_5 \neq 0 \wedge (X_4 \neq X_2 \wedge X_3 \neq X_1 \wedge X_1 \neq 5))$

3 Proposed contributions

Here I give the details of the novelty of my contribution.

3.1 (Generalized Decision) Diagrams

(Generalized Decision) Diagrams (GDDs) generalize EMDDs by allowing variable names (as well as constants) as edge labels. An (un-reduced) non-normalized GDD A is a finite directed edge-labeled tree of constant depth (K), encoding a function f_A from members $\langle x_K, x_{K-1}, \ldots, x_1 \rangle$ of the domain of natural tuples of length K ($\mathbb{N}^K$) to some finite range R, where each leaf node is a member of R, and each edge label, from a node at distance k from a leaf, is either: (1) a (Natural) constant, (2) a variable name from the set of variables $\{`x'_K, \ldots, `x'_1\} \setminus \{`x'_k, \ldots, `x'_1\}$, or (3) '*', and each edge from a given node is uniquely labeled, and each node has one edge labeled with '*'. An (un-reduced) *normalized* GDD is a GDD that also satisfies the rules listed in Section 3.1.3.

A *quasi-reduced GDD* (QGDD) is a directed acyclic graph that represents an un-reduced GDD, and has the additional property that there are no redundant nodes, that is, identical nodes have been collapsed to a common representation, so the resulting QGDD data structure contains no identical nodes. Fig. 6 shows sample GDD encodings.

Notation: Although we write x_k to refer to the value of the kth element of tuple x, we write 'x'$_k$ to refer to the name (used as a label) of the kth element of tuple x, hence 'x'$_k$ refers to the name 'x_3' when $k = 3$, but refers to a non-specific name when the value of k is not known. $labels(A)$ is the set of labels of edges from an node A, excluding '*' . For a non-leaf GDD (or QGDD) node A, we write $A\overrightarrow{p}$ to mean $p \in labels(A)$, and we write $A[p]$ for the node reached by the edge labeled p. Thus two nodes A and B are identical (written $A = B$) iff $labels(A) = labels(B) \wedge A[\text{'*'}] = B[\text{'*'}] \wedge \forall p \in labels(A) : A[p] = B[p]$. We write $level(A) = k$ iff the distance from A to a leaf node is k. We also write A to mean the entire tree at or below a GDD node A (or for all nodes reachable from an QGDD node A). We also write x_k for member k of a tuple x. GDD^K_k is the class of GDD nodes B where $level(B) = k$, encoding functions of K-tuples. With respect to a given level $k \leq K$, a K-tuple $x \in \mathbb{N}^K = \langle x_K, \ldots x_1 \rangle$ comprises a prefix $x_{\uparrow k} = \langle x_K, \ldots x_{k+1} \rangle \in \mathbb{N}^{K-k}$ and a suffix $x_{\downarrow k} = \langle x_k, \ldots x_1 \rangle \in \mathbb{N}^k$ and we say $x = x_{\uparrow k} x_{\downarrow k}$. We also impose a total ordering, $\succ$, on edge labels excluding '*', so that $\forall i, j \in \mathbb{N} : (i \succ j \text{ iff } i > j) \wedge (\text{'}x\text{'}_i \succ \text{'}x\text{'}_j \text{ iff } i > j) \wedge (i \succ \text{'}x\text{'}_j)$.

3.1.1 Decoding of GDDs

The meaning of a GDD A corresponds to the set of paths, from the root of A to leaves, whose edge labels match corresponding elements of the argument x of the function f_A encoded by A.

I first define the helper function *match* to indicate which edges match a given argument. I write $match(A, p, x)$ to mean that the edge labeled p from node A matches tuple x. Informally, $match(A, p, x)$ holds when the member, $x_{level(A)}$ of x has the value p (when $p \in \mathbb{N}$), or the value of the variable, within x, named by p, when p is a variable name.

I first define *eval* on labels other than '*' to simplify the definition of *match*. *eval* of an edge label p at level k also requires a tuple prefix (indicated as a subscript) used for evaluation of symbolic labels.

$$eval_{x\uparrow k}(\text{'}x\text{'}_i) = x_i \tag{1}$$

$$eval_{x\uparrow k}(c) = c \tag{2}$$

match has type: $\forall k, K | k \leq K : GDD^K_k \times (\mathbb{N} \cup \{\text{'}x\text{'}_K, \ldots, \text{'}x\text{'}_1\} \setminus \{\text{'}x\text{'}_k, \ldots, \text{'}x\text{'}_1\}) \times \mathbb{N}^K \rightarrow \{true, false\}$, and is defined as follows:

$$match(A, p, x) = A\overrightarrow{p} \wedge (eval_{x\uparrow k}(p) = x_k) \tag{3}$$

Thus, *match* requires A to have an outgoing edge label p that evaluates to x_k, in the context $x\uparrow k$.

The function f_A, of type $\mathbb{N}^K \rightarrow R \cup \{AmbiguityError\}$, encoded by a GDD A, where $level(A) = k$, is defined recursively as follows:

when $k = 0$, (A is a leaf node)

$$f_A(x) = A \tag{4}$$

when $k > 0 \wedge match(A, p, x)$ for exactly one edge label p, (A has one edge label matching x_k)

$$f_A(x) = f_{A[p]}(x) \tag{5}$$

when $k > 0 \wedge match(A, p, x) \wedge match(A, q, x) \wedge p \neq q$ for edge labels p, q, (multiple edges match x_k)

$$f_A(x) = AmbiguityError \tag{6}$$

otherwise (when $match(A, p, x)$ for no edge label p), (A has no edges matching x_k)

$$f_A(x) = f_{A[`*']}(x) \tag{7}$$

Thus, except when $AmbiguityError$ is encountered, the value of f at a non-leaf node is the value of f at one of its children, so that the value depends on the path, selected by members of the tuple, through the tree from the root to a leaf. Note that the choice of this path depends only on conditions of the forms: $x_k = c$ (for some constant c present in the GDD) and $x_k = x_i$ (for an index $i > k$), and their complements. Thus, for any given GDD, any two tuples which induce the same choices for such conditions will result in the in the choice of the same path.

3.1.2 Ambiguity rules for GDDs

Ambiguity in GDDs may occur due to the possibility of encountering $AmbiguityError$, and also when there are multiple ways to encode a label.

The first ambiguity rule avoids redundant label codes.

1. For each label $p \in labels(A)$ (so $p \neq$ '*'), the entire tree under $A[p]$ has no edges labeled 'x'$_k$, where $level(A) = k$.

   The label 'x'$_k$ can occur only within the tree rooted at A['*']. The label 'x'$_k$ is never needed in any tree rooted at $A[p]$, since the same effect could be obtained using the label p.

Thus the '*' label at level k acts as a kind of binder for 'x'$_k$, which may be thought of as a 'bound' variable at level k. We say 'x'$_k$ is a 'free' variable wherever it occurs within A['*'].

The $AmbiguityError$ of Equation 6 occurs when $eval_{x\uparrow k}(p) = eval_{x\uparrow k}(q)$ for two different non-'*' edge labels, p and q, from A. This can occur in two ways:

(1) $p = c$, for some constant $c \in \mathbb{N}$ and $x_k = c$, while $q =$ 'x'$_j$ for some $K \geq j > k$ and $x_j = c$, hence $x_k = x_j$. Thus a constant-labeled edge and a variable-labeled edge might both match the same argument. This ambiguity can be avoided by requiring $c \neq x_j$ to hold of the argument x of f_A

(2) $p =$ 'x'$_i$ for some $K \geq i > k$ and $x_k = x_i$, while $q =$ 'x'$_j$ for some $K \geq j > k$ and $x_k = x_j$, hence $x_k = x_i = x_j$. Thus two different variable-labeled edges might also both match the same argument. Similarly to the first case, this ambiguity can be avoided by requiring $x_i \neq x_j$ to hold of the argument x of f_A

The case where p and q are both constant labels never produces ambiguity because of the requirement that all edges from a given node have unique labels.

To ensure unambiguous definition of f, we associate, with each node A, a finite set C_A of inequality constraints, which will limit the domain of f_A. The constraints in C_A, where $level(A) = k$ are of 2 forms: (1) $c \neq x_j$ for some constant integer c and some $j > k$, and (2) $x_i \neq x_j$ for some $i > k$ and some $j > k$. We consider $p \neq q$ and $q \neq p$ to be the same constraint, for any labels p and q,

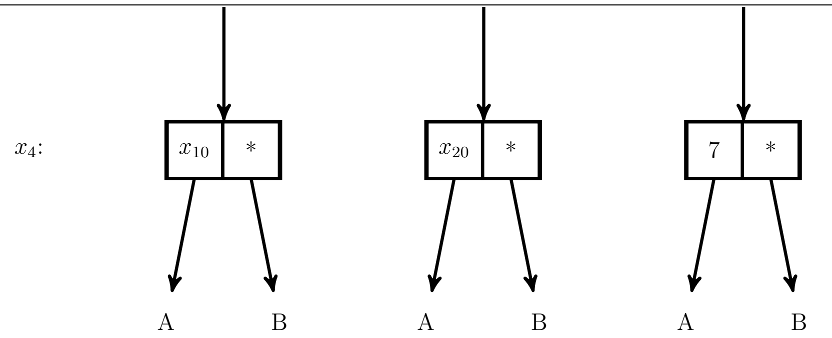
C_A is populated with exactly those constraints required by the following rules, which suffice to ensure the function f_A, encoded by GDD A, is well defined:

2. When $level(A) = 0$, $C_A = \emptyset$.

   No constraints are required to ensure $f_A \neq AmbiguityError$ in this case.

3. For each node A, index j and label p where $A\overrightarrow{p} \wedge A\overrightarrow{`x\text{'}_j} \wedge (p \neq `x\text{'}_j)$, we have $(p \neq x_j) \in C_A$.

   Two different edge labels must not become equivalent within the context of a given input. Thus, we constrain the input x so that Equation 6 will not immediately apply when applying f_A.



(a.) Suppose $x_{10} = 7$ (b.) Suppose $x_{20} = 7$ (c.)

Fig. 7: These GDDs at level 4 behave identically when $x_{10} = x_{20} = 7$ holds in the prefix.

4. For any node A with $level(A) = k$, and any labels p, q, r, we have $(A\overrightarrow{r} \vee r = \text{‘*’}) \wedge (p \neq q) \in C_{A[r]} \wedge p \neq \text{‘}x\text{’}_k \wedge q \neq \text{‘}x\text{’}_k \Rightarrow (p \neq q) \in C_A$.

   Constraints not involving ‘x’$_k$ are propagated from $C_{A[r]}$ to C_A.

5. For any node A with $level(A) = k$, and any label p, we have $(x_k \neq p) \in C_{A[\text{‘*’}]} \Rightarrow A\overrightarrow{p}$.

   Constraints $(x_k \neq p)$ involving ‘x’$_k$ are enforced at level k for A[‘*’] by requiring A to have a p-labeled edge.

Note that, for GDDs B with $level(B) = k$, these rules introduce constraints into C_B that involve only variables x_i with $i > k$, while such constraints are absorbed or transformed at level i, eliminating any constraints involving x_i from $C_{B'}$ (where $level(B') = i$), so that, when $level(A) = K$, $C_A = \emptyset$ always.

Thus, f_A, where A is a ‘top level’ GDD with $level(A) = K$, is always defined over the entire K-dimensional space of natural K-tuples, $\mathbb{N}^K$, while f_B, where B is a ‘lower level’ GDD with $level(B) = k < K$, is only defined for those members x of $\mathbb{N}^K$ which satisfy the constraints in C_B.

This suffices, because the recursive decoding process never uses f_B if the argument does not satisfy C_B. This is shown in [36](§7.48) Section A.1.

Since C_B involves only variables x_i with $i > k$, where $level(B) = k$, satisfaction of C_B by a tuple x, is actually a property of the prefix $x_{\uparrow k}$.

I will therefore write $sat(B)$ to mean the set of prefixes that satisfy the inequality constraints C_B. Thus, $x_{\uparrow level(A)} \in sat(A)$ implies x satisfies the inequality constraints C_A. I will also write $x \in sat(A)$ to mean the same thing.

An *unambiguous GDD* is a GDD that follows rules 1-5 above. Hereafter, all GDDs are unambiguous except where clearly noted.

3.1.3 Canonicity of GDDs

We say an encoding method E, decoded by a function f, is *canonical* over a domain D iff for each d in D there is only one encoding e in E such that $d = f_e$.

An additional rule involving equivalence of GDDs is needed to ensure canonicity of GDDs.

To prove canonicity, we must prove non-equivalence of non-identical GDDs. Certain prefixes, however, make some non-identical GDDs behave equivalently, as shown in Fig. 7.

I will define a form of equivalence on GDDs that avoids problematic prefixes.

The problem arises when a variable in the prefix corresponding to a variable name in one of the GDDs has the same value as a constant in one of the GDDs, or when two variables in the prefix corresponding to variable names in one of the GDDs have the same value.

I avoid this problem by considering only tuples where ‘free’ variables are chosen to avoid such coincidences. It suffices to use different large values for those variables, so these tuples (and prefixes) will be called *far-field tuples* (and *far-field prefixes*).

Thus, when considering equivalence of the GDDs in Fig. 7.a and Fig. 7.b, we would only consider prefixes where $x_{10} \neq x_{20}$, and when considering equivalence of the GDDs in Fig. 7.b and Fig. 7.c, we would only consider prefixes where $x_{20} \neq 7$.

Let us define, for a GDD A, the set of constants CO_A and the set of (indices of) *free variables* FV_A occurring in A.

Definition of CO and FV: When $A \in R$ is a leaf, $CO_A = \emptyset$, and $FV_A = \emptyset$.

Otherwise, $CO_A = \bigcup_{p \in labels(A) \cup \{\text{‘*’}\}} (CO_{A[p]}) \cup \{c | c \in \mathbb{N} \cap labels(A)\}$,

And $FV_A = (\bigcup_{p \in labels(A)} FV_{A[p]}) \cup (FV_{A[\text{‘*’}]} \setminus \{k\}) \cup \{i \in \mathbb{N} | \text{‘}x\text{’}_i \in labels(A)\}$, where $k = level(A)$

I also define, for a set S of GDDs (all having the same level), the set of far-field prefixes $far(S)$ for S.

$far(S) = \{\langle x_K \dots x_{k+1} \rangle \in \mathbb{N}^{K-k} | \forall i \in FV_S : \forall j \in FV_S \setminus \{i\} : \forall c \in CO_S : x_i \neq x_j \wedge x_i \neq c\}$, where $CO_S = \bigcup_{A \in S} CO_A$, and $FV_S = \bigcup_{A \in S} FV_A$, and k is the shared level for the GDDs in S.

As a special case, for a singleton $\{A\}$, we have:

$far(\{A\}) = \{\langle x_K \dots x_{k+1} \rangle \in \mathbb{N}^{K-k} | \forall i \in FV_A : \forall j \in FV_A \setminus \{i\} : \forall c \in CO_A : x_i \neq x_j \wedge x_i \neq c\}$, where $k = level(A)$.

Thus, in a far-field prefix for S, the value of each free variable never coincides with the value of any constant occurring in S or any other free variable. Since, for any GDD $A \in S$, C_A only has inequality constraints involving constants in CO_A, and variables in FV_A, they are automatically satisfied by any prefix in $far(S)$. Thus, $far(S) \subseteq sat(A)$, for every unambiguous GDD $A \in S$.

I also write $x \in far(S)$ when $x_{\uparrow k} \in far(S)$, in which case I say tuple x is far-field for S.

I now define *far-field equivalence* as follows:

$\forall A, B \in GDD_k^K, k \leq K :$

$A \underset{ff}{=} B$ iff: $\forall x \in \mathbb{N}^K \cap far(\{A, B\}) : f_A(x) = f_B(x)$.

For use in my canonicity rule, we need the following additional notation: I write $A - p$ (for an unambiguous GDD A and an edge label p), to mean a GDD A' where $labels(A') = labels(A) \setminus \{p\}$, and for every label q in $labels(A') \cup \{\text{‘*’}\}$, $A'[q] = A[q]$. Thus $A - p$ is a GDD node just like A except it is missing the edge labeled p and the subtree $A[p]$ under it. Note that $A - p$ is unambiguious iff $(\text{‘}x\text{’}_{level(A)} \neq p) \notin C_{A[\text{‘*’}]}$.

The following canonicity rules define *normalized* GDDs:

1. Rules 1-5 in Section 3.1.2 are satisfied, ensuring unambiguous definition of f.

2. For any node $A \in GDD_k^K$, and any label $p \in labels(A)$, we have either $(p \neq x_k) \in C_{A[\text{‘*’}]}$ or $\neg(A \underset{ff}{=} A - p)$.

   This rule forces normalized nodes to be ‘minimal’ in the sense that each outgoing edge, labeled $p \neq$ ‘*’, should have its existence justified, either by a constraint $(p \neq x_k)$ in $C_{A[\text{‘*’}]}$, which, by rule 5 requires the edge p to exist, or by the fact that A could not be effectively replaced by $A - p$.

In [36](§7.48) Sections A.2-A.3, I show (quasi-reduced) normalized GDDs, are canonical.

Henceforth all GDDs are assumed to be normalized (they follow these canonicity rules). When referring to possibly un-normalized diagrams, the symbol *UGDD* will be used.

3.1.4 Adequacy of GDDs

Choosing the range $R = \{true, false\}$ leads to encoding of boolean functions of tuples, which may be taken as characteristic functions of tuple sets. Thus tuple sets and tuple relations may be encoded by GDDs.

In [36](§7.48) Section A.4, I define *Bundles*, and the domain (*BundleUnions*) over which GDD-encoded functions apply, and show in [36](§7.48) Section A.5 that GDDs encode all Bundle-wise constant functions over BundleUnions, and that tuple-sets represented by GDD-encoded characteristic functions are closed over union, intersection, complement, and cartesian products.

GDDs encode all sets encoded by extensible QMDDs. This is fairly obvious, as QMDDs are similar to a special case of QGDDs, where no edge labels are variable names. In this case, ambiguity rules 1-5 in Section 3.1.2, (hence canonicity rule 1) are automatically satisfied with $C_A = \emptyset$ for all nodes A. Canonicity rule 2 remains unchanged.

GDDs also encode all sets encoded by FI MDDs, although this case is more complex. In lieu of a proof, note that, as with GDDs, each path through a FI-reduced MDD obviously corresponds to a Bundle [36](§7.48), so that all characteristic functions of sets encoded by FI-reduced MDDs are Bundle-wise constant.

GDDs are closed over complement, union, intersection, cartesian product, and product with the full set, due to the fact that they encode exactly all Bundle-wise constant functions.

3.1.5 Overhead improvement for symbolic methods

As noted in Section 2.7, there is overhead proportional to tuple size, when using unreduced decision diagrams to encode tuple-sets. It is desirable to find some way to decrease this overhead so that it is never onerous. A careful examination of the example in Section 2.7 shows that a constant-reduced encoding (where the constant is 1) reduces the BDD encoding of $\{\langle 1, 0, 1, 1, 0, 1\rangle\}$ to 2 non-leaf nodes, when null-pointer elimination is also used. Although such a reduction will typically produce a factor of 2 in space usage, it is desirable to encode sets using a number of nodes linear in the size of the set, so that DD encodings could, at least theoretically, compete with element list encodings.

The condition that makes a DD encoding (sometimes) occupy more nodes than the corresponding list encoding, is that some DD nodes have only a single (non-leaf) child. If every DD node of a K-level DD (encoding a set of K-tuples) had 2 children, the DD would have $2^K - 1$ nodes encoding a set of 2^K K-tuples, so that the overhead would be entirely reasonable. What is needed then, is a special encoding for a sequence of nodes having only a single child. Understanding my proposed solution to this problem requires noticing that, the explicit encoding is more compact only when many parts (or all) of the tuple may be encoded compactly as a string of a few bits (up to about the size of a pointer).

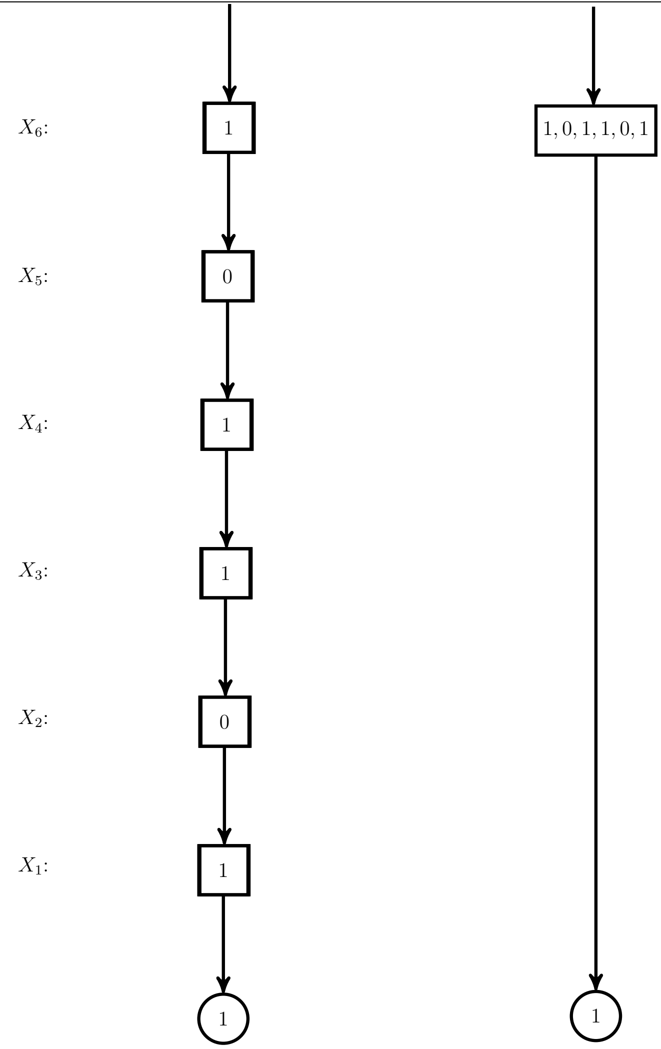
My proposed solution (for MDDs) is to condense a sequence of nodes having only a single (non-trivial) child into an additional annotation on an edge. Such an annotation contains a compact representation of the sequence of tuple element values associated with that sequence of nodes, as shown in Fig. 8. Thus, every node that has only one child is condensed to part of an annotation on an edge, so that all remaining nodes have at least 2 children each. This adjustment results in MDDs that encode sets with at least as many members as there are nodes in the encoding.

3.1.6 Reductions with GDDs

All the reductions mentioned in Section 2.1, and more, may be applied on a level-by-level basis to GDDs in a uniform manner, so that distinct algorithms to handle each reduction are no longer necessary. In each

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(a.) Explicit bit vector. (occupies only 64 bits)



(b.) Fully reduced BDD (implicit) with null-pointer elimination

(c.) Proposed solution with null-pointer elimination

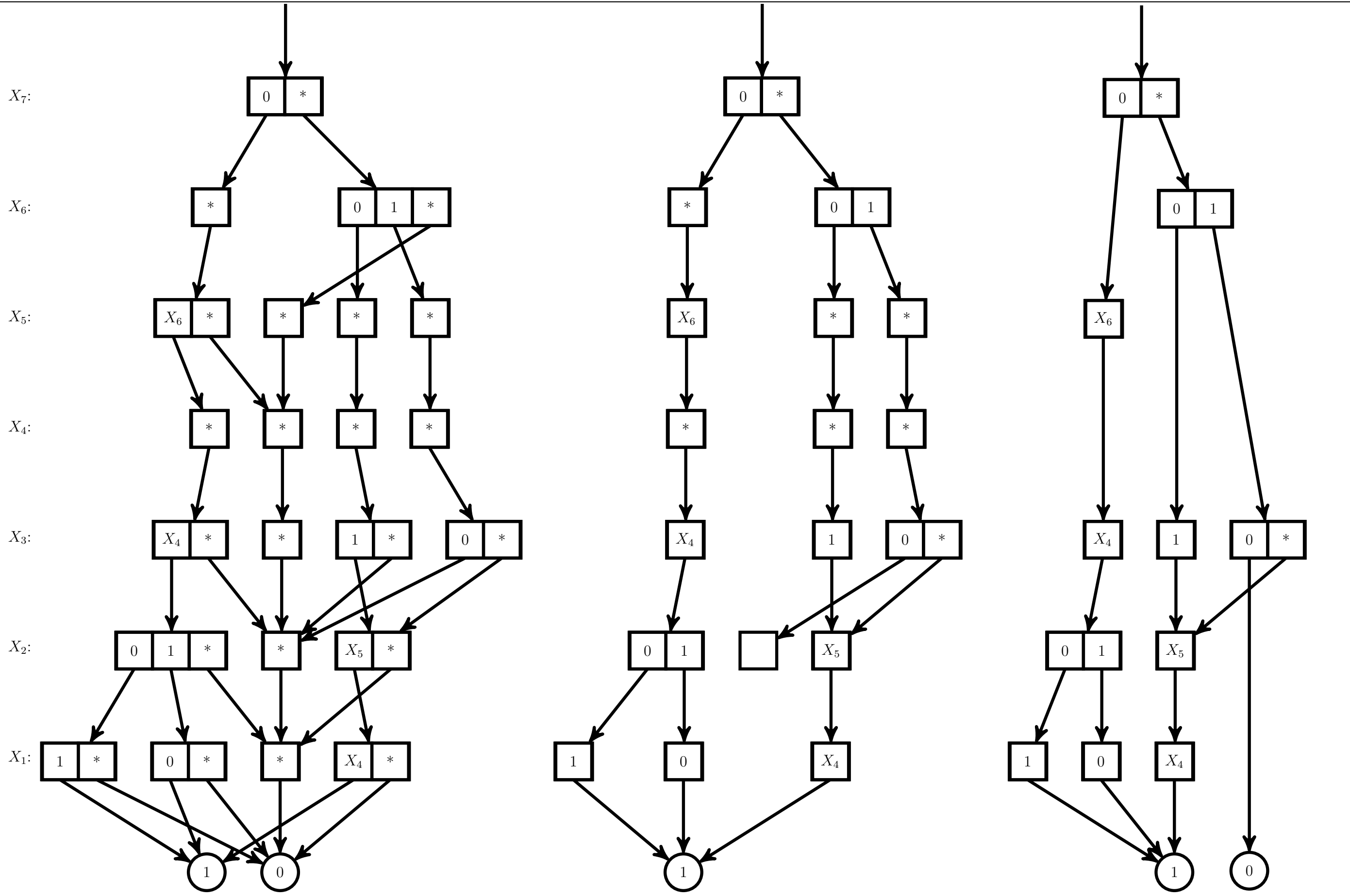
Fig. 8: Encodings of $\{\langle 1, 0, 1, 1, 0, 1\rangle\}$

case, an edge that skips a particular level is considered to be an abbreviated form for a very specific form of node, as follows:

1. Quasi-reduced is an essentially un-reduced form of DD not allowing skipped levels.

2. Fully-reduced level of a GDD. When level k is fully-reduced, an edge from node A which skips level k and leads to a node C at level $j < k$ is equivalent to the same edge from node A leading instead to a node B at level k, where B has a single edge leading to C, labeled '$*$'. Thus, a quasi-reduced GDD can be converted to a fully-reduced GDD by skipping nodes having only the '*'-labeled edge, as shown in Fig. 6.

3. Identity-reduced level of a GDD. When level k is Identity-reduced, an edge from node A which skips level k and leads to a node C at level $j < k$ is equivalent to the same edge from node A leading instead to a node B at level k, where B has 2 edges, one labeled 'X_{k+1}' leading to C, and the other labeled '$*$' leading to an encoding of the empty set.



(a.) Quasi-reduced GDD

(b.) Quasi-reduced GDD with null pointer elimination

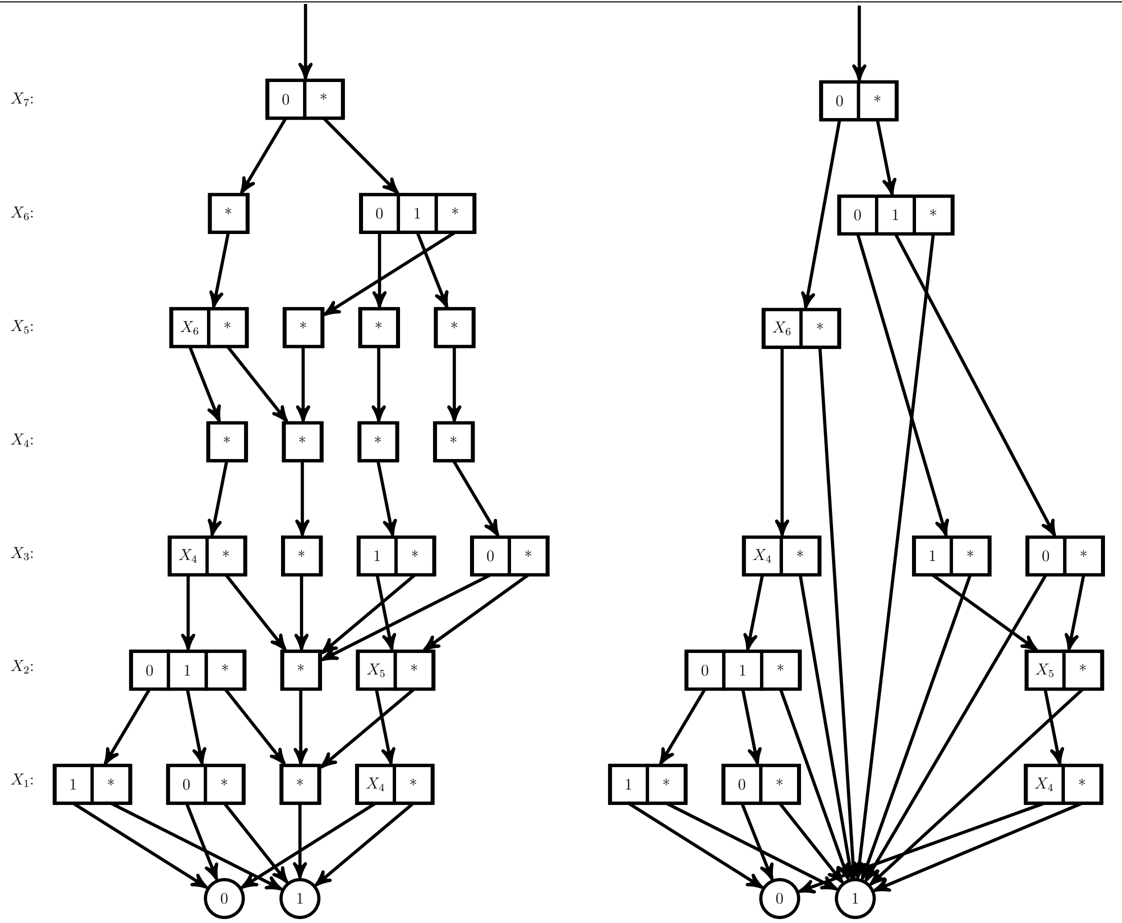
(c.) Fully-reduced GDD with null ptr elimination

Fig. 9: Encodings of $f(X) = (X_7 = 0 \wedge X_5 = X_6 \wedge X_3 = X_4 \wedge (X_2 = 0 \wedge X_1 = 1 \vee X_2 = 1 \wedge X_1 = 0)) \vee (X_7 \neq 0 \wedge X_2 = X_5 \wedge X_1 = X_4 \wedge (X_6 = 0 \wedge X_3 = 1 \vee X_6 = 1 \wedge X_3 \neq 0))$

4. Constant-reduced level of a GDD. When level k is Constant(c)-reduced, an edge from node A which skips level k and leads to a node C at level $j < k$ is equivalent to the same edge from node A leading instead to a node B at level k, where B has 2 edges, one labeled 'c' leading to C, and the other labeled '$*$' leading to an encoding of the empty set.

For purposes of maintaining canonicity, obviously the 'equivalent' node B must never exist, as that would create multiple encodings for a given function. A Fully-reduced level of a GDD, for example, must never have a node with only a single edge labeled '$*$', since a path through such a node should simply skip that level.

The use of the 'empty set', meaning a GDD encoding of a function that always returns 0, in items 3 and 4 points out the asymmetric nature of null-pointer elimination. The null pointer customarily is used to encode a function that always returns 0, but there is no such simple customary encoding for a function that always returns a constant other than 0. This situation is incompatible with QGDDs and is illustrated in Fig. 9(b.) where the GDD from Fig. 9(a.) has been modified by null pointer elimination, but is not fully reduced. Although there is appreciable reduction in nodes and edges, the necessity of including an awkward empty node indicates that something is wrong. The awkward situation is relieved in Fig. 9(c.) where full reduction is applied. The complement operation, which nominally can be performed by exchange



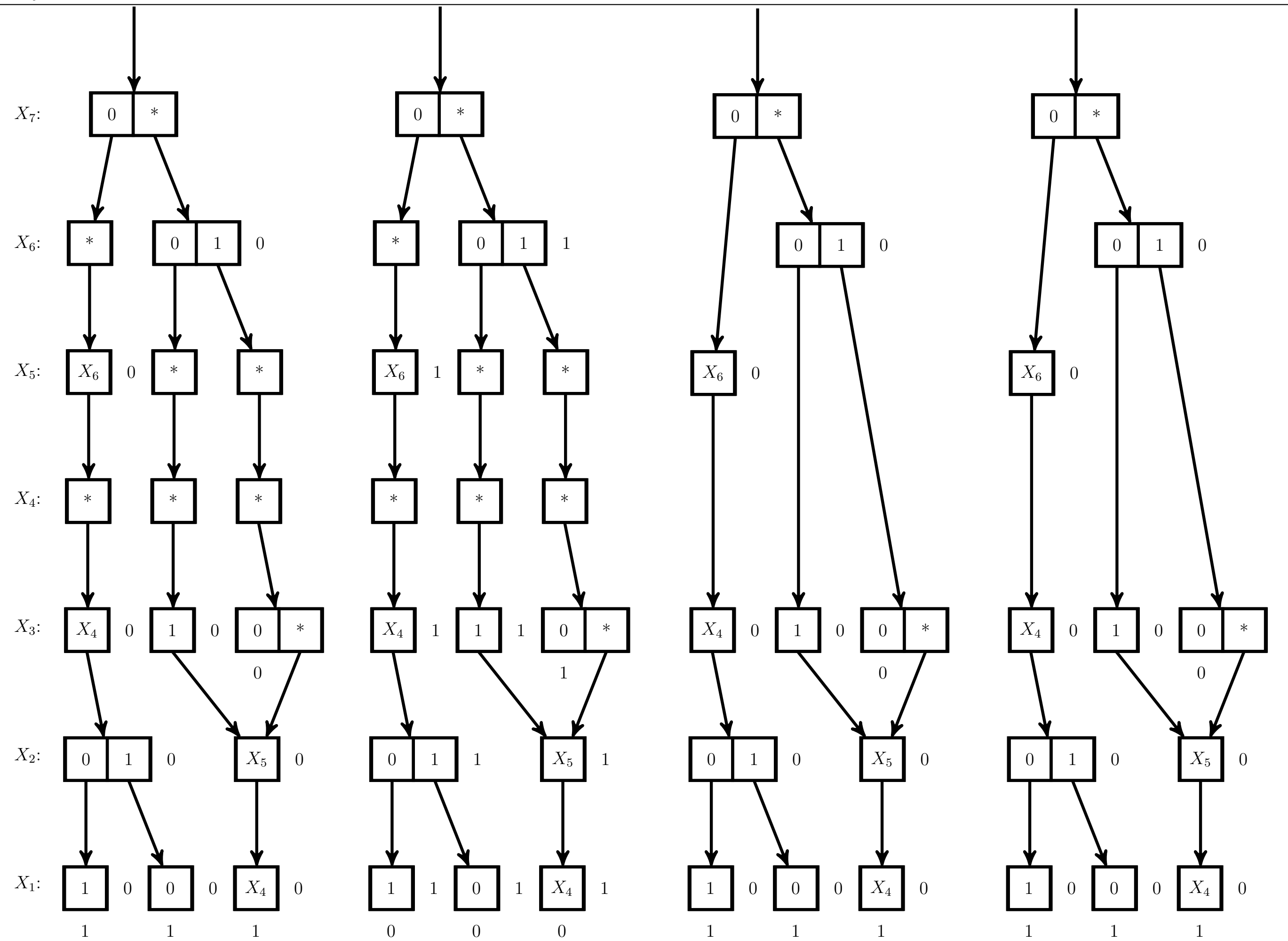
(a.)(b.) Quasi-reduced GDD
with or without null pointer elimination

(c.) Fully-reduced GDD
with or without null pointer elimination

Fig. 10: Encodings of $f(X) = \neg((X_7 = 0 \wedge X_5 = X_6 \wedge X_3 = X_4 \wedge (X_2 = 0 \wedge X_1 = 1 \vee X_2 = 1 \wedge X_1 = 0)) \vee (X_7 \neq 0 \wedge X_2 = X_5 \wedge X_1 = X_4 \wedge (X_6 = 0 \wedge X_3 = 1 \vee X_6 = 1 \wedge X_3 \neq 0)))$. This function is the negation of the function encoded in Fig. 9, so diagram (a.) is derived from the diagram in Fig. 9(a.) by exchanging the two leaf labels.

of 'true' and 'false' leaves, must, in practice include special code to process cases involving encodings of the empty set or of the 'full' set. This asymmetry is seen by comparing the GDD for the complement in Fig. 10(b.) with Fig. 9(b.), or by comparing the fully reduced complement GDD in 10(c.) with Fig. 9(c.). I expect to remedy this asymmetry in my GDD algorithm library by providing *symmetric leaf elimination*, a simple encoding for all functions that always return a (Boolean) constant, while eliminating the need for the constant leaves. Use of such an encoding will be indicating by writing the constant just below the box containing the edge label, as shown in Fig. 11. Additionally, if the edge label is '*', the '*' will be omitted, and the constant will be written to the right of the node. Fig. 11(a.) and (b.) show that symmetric leaf elimination condenses the encoded form of the function from Fig. 9 and its negation equally, and that also holds true for the fully reduced encoding (c.) and its negation (d.) as well.

With the use of diagrams with symmetric leaf elimination, the previously mentioned reductions may now be illustrated as in Fig. 12(a.). Consideration of the previous examples makes obvious the need for the additional reductions in Fig. 12(b.), which complement the existing set.



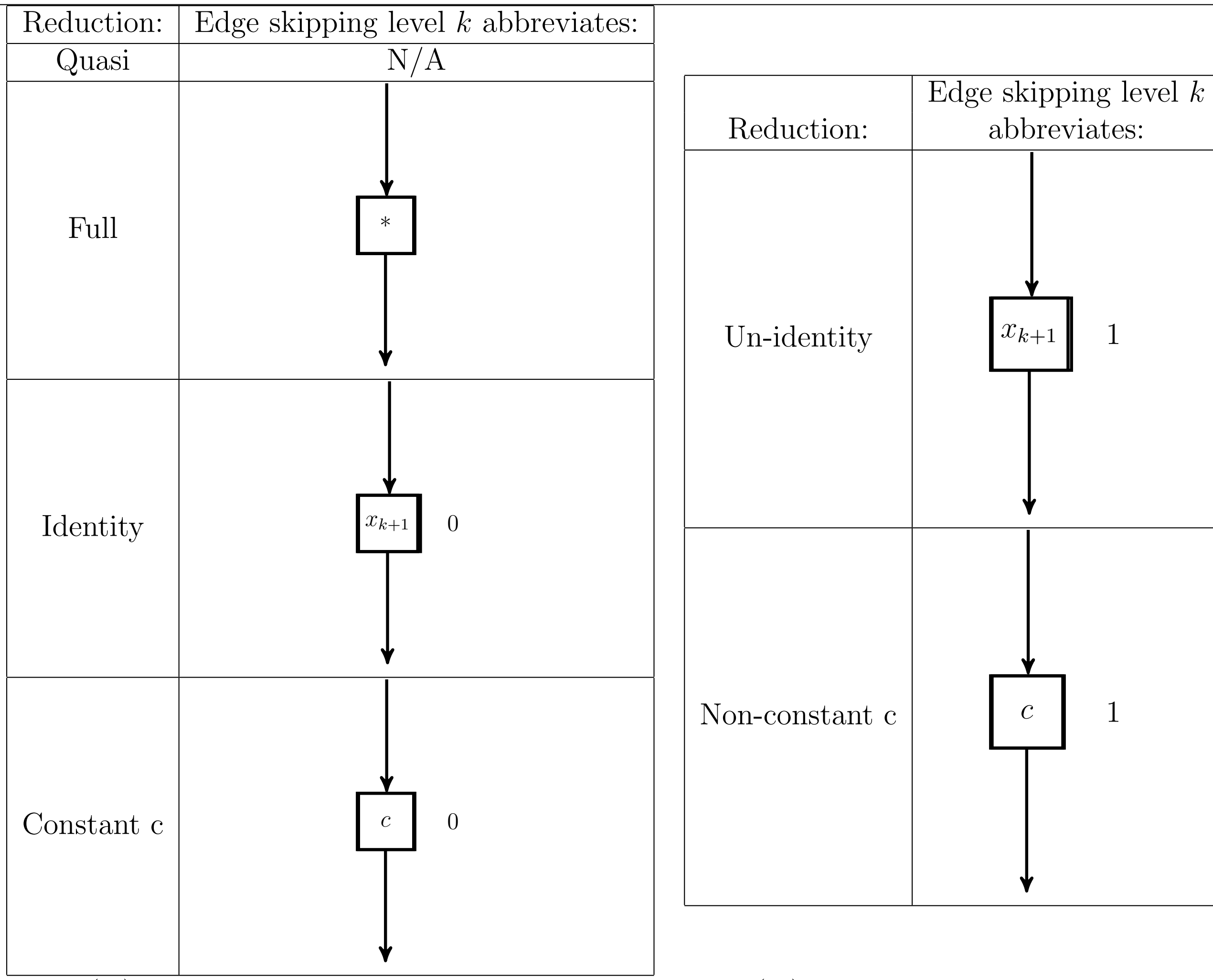
(a.) Quasi-reduced GDD (b.) Complement of (a.) (c.) Fully-reduced GDD (d.) Complement of (c.)

Fig. 11: Encodings of $f(X) = (X_7 = 0 \wedge X_5 = X_6 \wedge X_3 = X_4 \wedge (X_2 = 0 \wedge X_1 = 1 \vee X_2 = 1 \wedge X_1 = 0)) \vee (X_7 \neq 0 \wedge X_2 = X_5 \wedge X_1 = X_4 \wedge (X_6 = 0 \wedge X_3 = 1 \vee X_6 = 1 \wedge X_3 \neq 0))$ and its negation with symetric leaf elimination

3.1.7 Automatic reduction selection per edge

One disadvantage with the use of various reduction schemes is that, in current DD libraries, the reduction scheme for each DD must be chosen by the modeler using knowledge of the expected structure of the encoded set. Typically, an entire DD must be encoded using a single reduction scheme, so that different branches of a DD must be encoded using the same reduction scheme, although the scheme may vary by level, as with fully-identity reduction, where full reduction is applied to even levels, and identity reduction is applied to odd levels.

The GDD from Fig. 11 is shown in Fig. 13 with 4 different reductions applied. It can be seen that each reduction has advantages for some part of the GDD, and that in general, advantageous selection of the appropriate reduction requires the modeler to have deep understanding of the encoding of sets and of the sets themselves. Thus, high-performance model checking tools require domain specific knowledge in their construction. This situation is not reasonable for researchers attempting to build general purpose tools. However, the necessity of manual selection of reductions may be removed, and the flexibility of



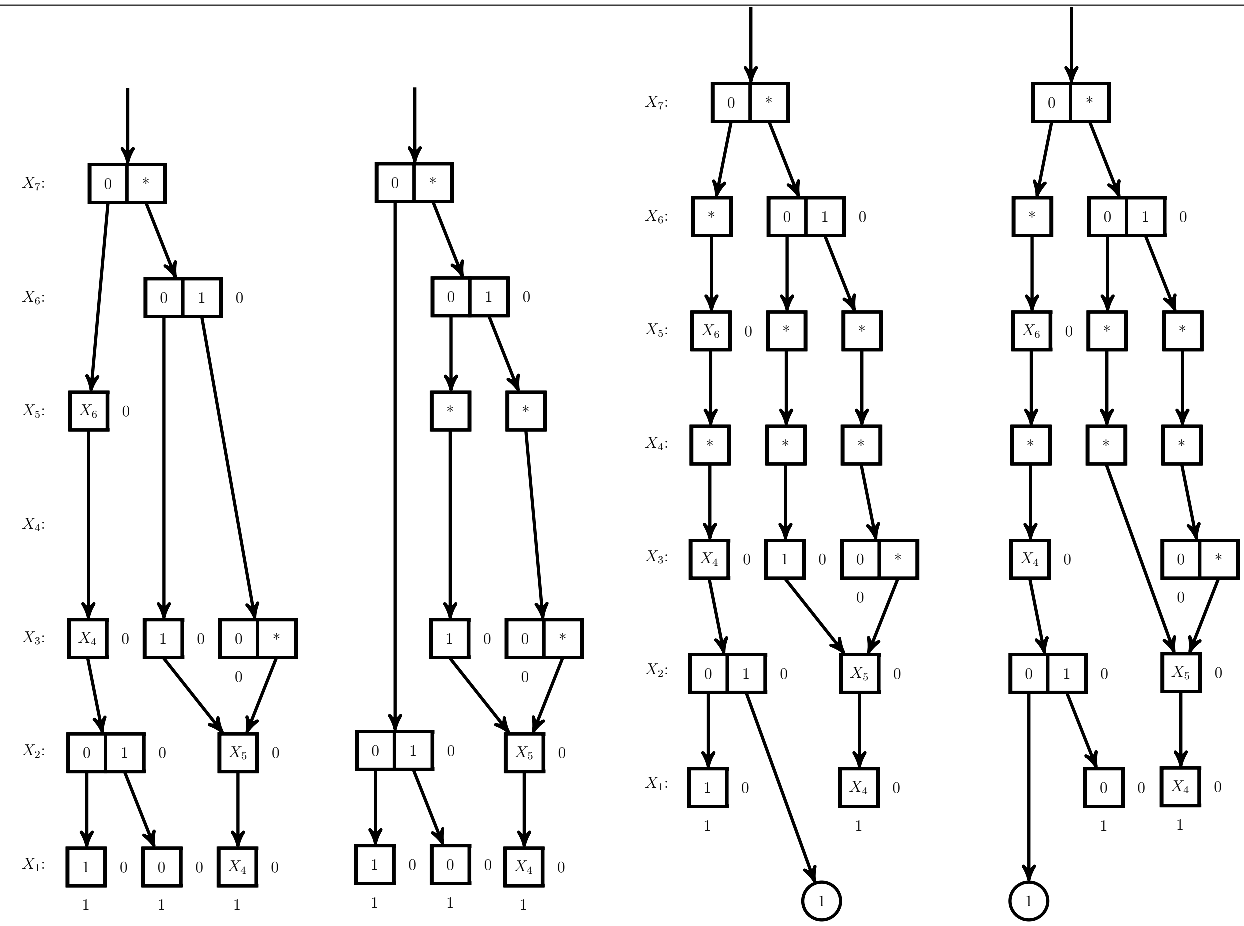
(a.) Previously mentioned reductions (b.) Suggested new reductions

Fig. 12: Reductions use skipped levels as abbreviations for certain nodes.

branch-specific reductions may be added, through an adjustment of the scheme described in Section 3.1.5.

In this scheme, within a GDD, certain very simple nodes, such as those which would be drawn with only 1 box, would be condensed to part of an edge annotation, and such annotations would be additionally condensed in cases where a simple pattern repeats many times, as would be reduced in fully or fully-identity reduced GDDs. Fig. 14(a.) shows the application of this scheme to reduction of the Quasi-reduced GDD from Fig. 11(a.), while Fig. 14(c.) is a legend listing the node abbreviations used in the edge annotations of Fig. 14(a.) and (b.).

This scheme resembles the scheme of Section 3.1.5, with the adjustment that the edge annotations describe a brief sequence of reductions, instead of a sequence of small indices. To understand this reduction scheme, consider the first (leftmost) node label box of the root node in Fig. 14(a.). The edge label (0) is followed by the annotation: '$[F, I]^2$', which is interpreted according to Fig. 14 to be the sequence of nodes: $\boxed{*}$, $\boxed{x_{k+1}}$ 0, $\boxed{*}$, $\boxed{x_{k+1}}$ 0, that is, the edge is considered to be reduced by full reduction at level 6, identity reduction at level 5, full reduction at level 4 and identity reduction at level 3, thus corresponding to the actual nodes found in Fig. 11(a). Each level-skipping edge has an annotation that is customized for that edge and the removed nodes it encodes. Thus, such an encoding scheme has the



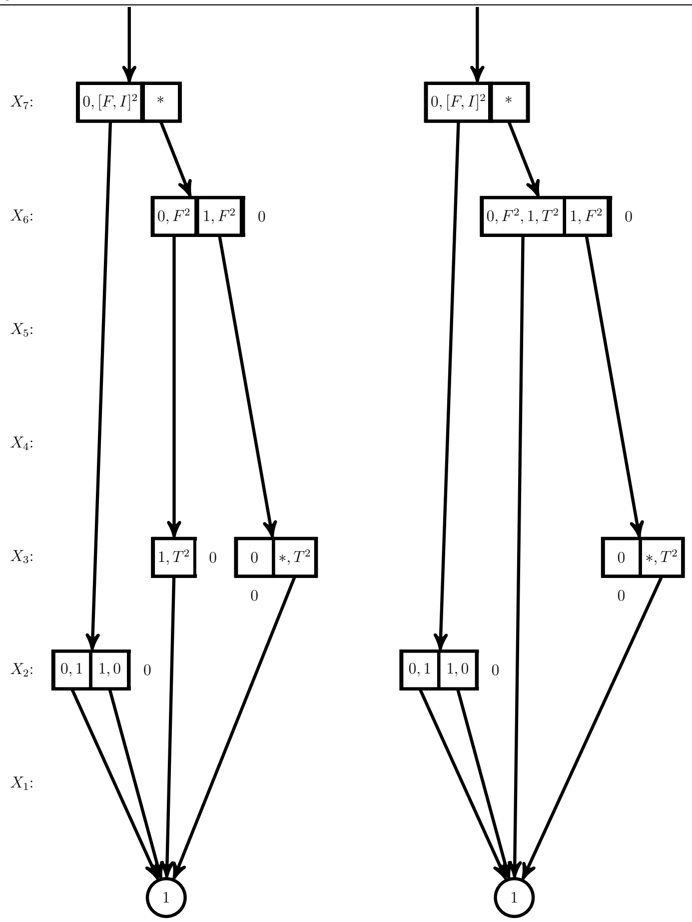
(a.) Fully-reduced GDD (b.) FI-reduced (c.) Constant 0-reduced (d.) Const. 1-reduced

Fig. 13: Encodings of $f(X) = (X_7 = 0 \wedge X_5 = X_6 \wedge X_3 = X_4 \wedge (X_2 = 0 \wedge X_1 = 1 \vee X_2 = 1 \wedge X_1 = 0))$ $\vee(X_7 \neq 0 \wedge X_2 = X_5 \wedge X_1 = X_4 \wedge (X_6 = 0 \wedge X_3 = 1 \vee X_6 = 1 \wedge X_3 \neq 0))$ using various reductions

benefit of automatic condensation of GDD nodes when any of the previously discussed reductions would apply, without the necessity of manual selection. Thus, this feature provides the benefits of all extant reduction techniques, without additional analysis effort on the part of the library user.

In this diagram I have been careful to limit each edge annotation to a small amount of information. Specifically, an edge annotation is limited to either a repeated pattern of a few reductions, as in the root node in Fig. 14(a.), or a list of a few small constants, as with the node at level 2. In such cases, it appears that the benefit of this reduction scheme outweighs the cost of the annotation size. If an edge annotation were unlimited in size, allowing the description of a lengthy sequence of reduction types, the same GDD would be reduced as shown in Fig. 14(b.). This diagram has one fewer node, but it also has a rather complex annotation on the first edge from the node at level 6. The consequences for this may be that the data structure representing edges would be larger for all edges, resulting in a net loss for diagrams where there are few opportunities for reduction.

In more distant future work, an additional steps along these lines could be taken, however, schedule limitations may place such work beyond the scope of the proposed research.



symbol	reduction	node
$[A]^n$	$A, \ldots, A$ (n times)	
0	constant 0	0 0
1	constant 1	1 0
F	full	$*$
I	identity	x_{k+1} 0
T	identity'	x_{k+3} 0

(a.) GDD cleverly reduced (b.) Excessively reduced (c.) Edge annotation legend

Fig. 14: Encodings of $f(X) = (X_7 = 0 \wedge X_5 = X_6 \wedge X_3 = X_4 \wedge (X_2 = 0 \wedge X_1 = 1 \vee X_2 = 1 \wedge X_1 = 0)) \vee (X_7 \neq 0 \wedge X_2 = X_5 \wedge X_1 = X_4 \wedge (X_6 = 0 \wedge X_3 = 1 \vee X_6 = 1 \wedge X_3 \neq 0))$ using automatic reduction scheme

3.1.8 Algorithms for GDDs

Algorithms for GDD manipulation are generally expected to resemble those for MDD manipulation except for the following characteristic. Meanings of subgraphs of GDDs obviously have more context dependence than meanings of subgraphs of MDDs. This dependence causes non-trivial operations (such as set union) to have multiple values when they are implemented in a context-independent manner. Each result value is valid in certain contexts associated with possible sets of constraints on 'higher' variables not accessed by the operation. I anticipate that all of the multiple values of an operation will be cached for later use.

Due to time constraints I previously deferred the writing of efficient algorithms until after the initial version of this proposal. However, since the proposed schedule has me working on the algorithm library since the start of July, I included an algorithm for union of GDD-encoded sets in [36](§7.48).

3.2 TeDDy interface improvement through C++ metaprogramming

In lieu of a formal description of the details of the proposed improvements, consider the following algorithm, as published in [38](§7.3):

“

MDD $Bisim$(level k, MDD $\overline{\mathcal{T}}_{\mathcal{E}}$, MDD $\mathcal{B}_{in}$) is • *memoize this function*
local MDD $\mathcal{B}$, MDD $\mathcal{B}_{old}$, MDD $\mathcal{Z}$, MDD $\mathcal{Z}^R$, MDD $\mathcal{D}$, MDD $\mathcal{D}^R$
if $k = 0$ then return $\mathcal{B}_{in}$; • *leaf case*
$\mathcal{B} \leftarrow \mathcal{B}_{in}$;
$\mathcal{B} \leftarrow BisimSaturate(k, \overline{\mathcal{T}}_{\mathcal{E}}, \mathcal{B})$; • *saturate below this level*
repeat
  $\mathcal{B}_{old} \leftarrow \mathcal{B}$;
  for each $e \in \mathcal{E}$ where $Top(\ \overline{\mathcal{T}}_e\ ) = k$, loop • *let* $p \xrightarrow{\overline{e}} p' \triangleq \langle p, p' \rangle \in \overline{\mathcal{T}}_e$
    $\mathcal{Z} \leftarrow \{(p', q) | \exists q' : q \xrightarrow{\overline{e}} q' \wedge p'\mathcal{B}q'\}$; • *in 1 symbolic step (composition)*
    $\mathcal{Z}^R \leftarrow \{(p, q') | \exists p' : p \xrightarrow{\overline{e}} p' \wedge p'\mathcal{B}q'\}$; • *also 1 step composition*
    $\mathcal{D} \leftarrow \{(p, q) | \exists p' : (q \in \mathcal{S}_k \times \ldots \times \mathcal{S}_1) \wedge (p \xrightarrow{\overline{e}} p') \wedge \neg(p'\mathcal{Z}q)\}$; • *also 1 step*
    $\mathcal{D}^R \leftarrow \{(p, q) | \exists q' : (p \in \mathcal{S}_k \times \ldots \times \mathcal{S}_1) \wedge (q \xrightarrow{\overline{e}} q') \wedge \neg(p\mathcal{Z}^R q')\}$; • *also 1 step*
    $\mathcal{B} \leftarrow \mathcal{B} \setminus (\mathcal{D} \cup \mathcal{D}^R)$; • *refine*
    $\mathcal{B} \leftarrow BisimSaturate(k, \overline{\mathcal{T}}_{\mathcal{E}}, \mathcal{B})$; • *re-saturate*
  end loop;
until $\mathcal{B}_{old} = \mathcal{B}$;
return $\mathcal{B}$;

Fig. 2. Bisimulation algorithm using Saturation heuristic.

MDD $BisimSaturate$(level k, MDD $\overline{\mathcal{T}}_{\mathcal{E}}$, MDD $\mathcal{B}$) is
local MDD $\mathcal{B}'$, MDD $\mathcal{B}_*$, MDD $\mathcal{B}'_*$
$\mathcal{B}' \leftarrow$ new empty mutable MDD node;
for each index L in $\mathcal{B}$ loop • *loop over children*
  $\mathcal{B}_* \leftarrow \mathcal{B}_L$;
  $\mathcal{B}'_* \leftarrow$ new empty mutable MDD node;
  for each index R in $\mathcal{B}_*$ loop • *loop over grandchildren*
    $\mathcal{B}'_{*R} \leftarrow Bisim(k-1, \overline{\mathcal{T}}_{\mathcal{E}}, \mathcal{B}_{*R})$;
  end loop;
  $\mathcal{B}'_L \leftarrow unique(\mathcal{B}'_*)$;
end loop;
return $unique(\mathcal{B}')$;

Fig. 3. Helper function for Bisimulation algorithm.

”

My saturation-based bisimulation algorithm, exactly as printed in [38](§7.3).

The first C++ version of the code inside *Bisim* was (after removing the most embarrassing parts):

// Parameters: int L, MDDL::mdd hatS, MDDL::mdd * T, int NQ, MDDL::mdd Bin, MDDL::OperatorCode cache

```
// calculate T2 and NQ2, for lower levels
MDDL::mdd * T2 = T;
int NQ2 = NQ;
const int hatSL = hatS.level();
const int SXSL = hatSL+hatSL;
while( NQ2 && (T2[0].level()==SXSL)){ T2++; NQ2--; };
//convenience:
int L3 = L+L+L;
// utility stuff:
bool * oddlevels = levelsmod( levelsmod<bool>( NULL, 4*L, false, 1, 0), 4*L, true, 2, 1 ); oddlevels[0]=false;
bool * alllevels = levelsmod<bool>( NULL, 4*L, true, 1, 0); alllevels[0]=false;
// mod 3 utility stuff:
bool * levels1 = levelsmod( levelsmod<bool>( NULL, 3*L, false, 1, 0), 3*L, true, 3, 1 ); levels1[0]=false; // variable 3rd of 3
bool * levels2 = levelsmod( levelsmod<bool>( NULL, 3*L, false, 1, 0), 3*L, true, 3, 2 ); levels2[0]=false; // variable 2nd of 3
bool * levels3 = levelsmod( levelsmod<bool>( NULL, 3*L, false, 1, 0), 3*L, true, 3, 0 ); levels3[0]=false; // variable 1st of 3
bool * levels12 = levelsmod( levelsmod( levelsmod<bool>( NULL, 3*L, false, 1, 0), 3*L, true, 3, 1 ), 3*L, true, 3, 2 ); levels12[0]=false;
bool * levels23 = levelsmod( levelsmod( levelsmod<bool>( NULL, 3*L, false, 1, 0), 3*L, true, 3, 2 ), 3*L, true, 3, 0 ); levels23[0]=false;
bool * levels13 = levelsmod( levelsmod( levelsmod<bool>( NULL, 3*L, false, 1, 0), 3*L, true, 3, 1 ), 3*L, true, 3, 0 ); levels13[0]=false;
int * sig2Lop = levelsmod( levelsmod( levelsmod<int>( NULL, 3*L, (int)0x88888888, 1, 0), 3*L, (int)0xAAAAAAAA, 3, 0), 3*L, (int)0xCCCCCCCC, 3, 1); sig2Lop[0]=0x88888888;//...
int * sig2Luse = levelsmod( levelsmod( levelsmod<int>( NULL, 3*L, 3, 1, 0), 3*L, 1, 3, 0), 3*L, 2, 3, 1); sig2Luse[0]=3; //variable1: 0; variable2: 1,0; variable3: 1;
int * sig2Rop = levelsmod( levelsmod( levelsmod<int>( NULL, 3*L, (int)0x88888888, 1, 0), 3*L, (int)0xAAAAAAAA, 3, 0), 3*L, (int)0xCCCCCCCC, 3, 2); sig2Rop[0]=0x88888888;//...
int * sig2Ruse = levelsmod( levelsmod( levelsmod<int>( NULL, 3*L, 3, 1, 0), 3*L, 1, 3, 0), 3*L, 2, 3, 2); sig2Ruse[0]=3; //variable1: 0; variable2: 1; variable3: 1,0;
int * DeltaBbarop = levelsmod<int>( NULL, 3*L, (int)0xFFFFFFFE, 1, 0); DeltaBbarop[0]=0x4F444F44; //variable1: 3|1; variable2: 3|1; variable3: 3|1; variable 0: (3&!2)|(1&!0);
int * DeltaBbaruse = levelsmod( levelsmod( levelsmod<int>( NULL, 3*L, 15, 1, 0), 3*L, 11, 3, 0), 3*L, 14, 3, 2); DeltaBbaruse[0]=15; //variable1: 3,1,0; variable2: 3,2,1,0...
int * DeltaBbaralt2use = levelsmod( levelsmod( levelsmod<int>( NULL, 3*L, 15, 1, 0), 3*L, 15, 3, 0), 3*L, 15, 3, 1); DeltaBbaralt2use[0]=15; //variable2: 3,2,1,0 everywhere;

MDDL::mdd B = Bin;
MDDL::mdd Bold = B;

// first, saturate lower levels:
B = BiSat1Saturation( L, hatS, T2, NQ2, B, cache );

// then do fixed point calculation:
do {
    Bold = B;
    // operate at this level:
    // apply each transition relation at this level (T is sorted by descending Top level):
    for ( int a=0; (a<NQ) && (T[a].level() == SXSL); ++a ) {
        MDDL::mdd Ta = T[a];

        // sig1L = (B X hatS) & (T X2 hatS) // actually just (T X2 hatS)
        MDDL::mdd sig1L = InsertDontCaresQQ( L3, Ta, levels2, hatS, MDD_INSERTDC5_QQ );

        // sig1R = (B X hats) & (hats X T) // actually just (hats X T)
        MDDL::mdd sig1R = InsertDontCaresQQ( L3, Ta, levels3, hatS, MDD_INSERTDC3_QQ );

        // sig2L = (B o T) X2 hatS // actually just (B o T)
        // sig2Lpre = (B o T)pre = (* X B) & (T X *)
        MDDL::mdd sig2Lpre = GenericCompose4QQ( 0, 0, B, Ta, sig2Luse, sig2Lop, L3, MDD_GCOMPOSE43_QQ );
        MDDL::mdd sig2L = ProjectUnionQQ( sig2Lpre, levels2, MDD_PROJECTU5_QQ );
        MDDL::mdd sig2Lalt = InsertDontCaresQQ( L3, sig2L, levels2, hatS, MDD_INSERTDC5_QQ );

        // sig2R = hatS X (Binv o T) // actually just (Binv o T)
        // sig2Rpre = (Binv o T)pre = (* X B) & (T X2 *)
        MDDL::mdd sig2Rpre = GenericCompose4QQ( 0, 0, B, Ta, sig2Ruse, sig2Rop, L3, MDD_GCOMPOSE44_QQ );
        MDDL::mdd sig2R = ProjectUnionQQ( sig2Rpre, levels1, MDD_PROJECTU4_QQ );
        MDDL::mdd sig2Ralt = InsertDontCaresQQ( L3, sig2R, levels3, hatS, MDD_INSERTDC3_QQ );

        // DeltaBbarpre = sig1L\(hatS X sig2R) U sig1R\(sig2L X2 hatS)
        MDDL_ASSERT((hatS));
        MDDL::mdd DeltaBbarpre = GenericCompose4QQ( sig1L, sig2R, sig1R, sig2L, DeltaBbaruse, DeltaBbarop, L3, MDD_GCOMPOSE45_QQ );
        MDDL::mdd DeltaBbarpre1alt = MDDL::g_mddf.minus_qq(sig1R,sig2Lalt);
        MDDL::mdd DeltaBbarpre2alt = MDDL::g_mddf.minus_qq(sig1L,sig2Ralt);
        MDDL::mdd DeltaBbarprealt = MDDL::g_mddf.or_qq(DeltaBbarpre2alt,DeltaBbarpre1alt);

        MDDL::mdd DeltaBbar = ProjectUnionQQ( DeltaBbarpre, levels1, MDD_PROJECTU4_QQ );

        // put the results into B
        B = MDDL::g_mddf.minus_qq( B, DeltaBbar );

        // saturate those results:
        {if(B) B = BiSat1Saturation( L, hatS, T2, NQ2, B, cache ); };
        //MDDL_ASSERT((B));
    };
} while ( Bold != B );

// destruct utilities
CDEL(DeltaBbaruse);
CDEL(DeltaBbarop);
CDEL(sig2Ruse);
CDEL(sig2Rop);
CDEL(sig2Luse);
CDEL(sig2Lop);
CDEL(levels13);
CDEL(levels23);
CDEL(levels12);
CDEL(levels3); CDEL(levels2); CDEL(levels1);
CDEL(alllevels); CDEL(oddlevels);

MDDL::g_mddf.cache_add( cache, Bin, B );  // memoize
return B;
```

Note the amorphous block of code near the beginning, containing many calls to the levelsmod function. That code calculates the MDD equivalent of ‘dope vectors’, used in later operations to indicate choices associated with individual MDD levels of operands in later MDD operations. Those operations are mostly set and relational compositions of various kinds within the nested do-while and for loops. These calls are located above the loop, as the vectors are independent of the loop iteration in which they are used. Generation of this code is very error-prone and tedious. I made a number of templates to simplify this code specifically, so that generation of the vectors now uses code that has some noticeable relation to the compositional operations being performed. The next version was:

Parameters: int L, MDDL::mdd hatS, MDDL::mdd * T, MDDL::mdd * Tinv, int NQ, MDDL::mdd Bin, MDDL::OperatorCode cache

```
// calculate T2, T2inv, and NQ2, for lower levels
MDDL::mdd * T2 = T;
MDDL::mdd * T2inv = Tinv;
int NQ2 = NQ;
const int hatSL = hatS.level();
const int SXSL = hatSL+hatSL;

while( NQ2 && (T2[0].level()==SXSL)){ T2++; T2inv++; NQ2--; };

// variables for evaluating B\((U2/3 (T X hatS)\(_ X BoTinv)) U (U2/3 (hatS X Tinv)\(ToB X _)))
// == B\(  {(p q)| exists p': (p T p') /\ (q in hatS) /\ ~(exists q': p' B q' /\ q' Tinv q)}
//       U {(p q)| exists q': (p in hatS) /\ (q' Tinv q) /\ ~(XXXexists p': p B p' /\ p' Tinv q'XXX)})
static Symbolic_Variable p("p",L);
static Symbolic_Variable pprime("pprime",L);
static Symbolic_Variable q("q",L);
static Symbolic_Variable qprime("qprime",L);

// B o Tinv // == {(p' q)| exists q': p' B q' /\ q' Tinv q}
static composition_spec OP_BoTinv(*this); // should be same as OP_ToB
OP_BoTinv.initialize2( (bool1&bool0), ( pprime % qprime % q ), USE()<< pprime << qprime, USE()<< qprime << q, PROJECT()<< qprime );

// T o B // == {(p q')| exists p': p T p' /\ p' B q'}
static composition_spec OP_ToB(*this); // should be same as OP_BoTinv
OP_ToB.initialize2( (bool1&bool0), ( p % pprime % qprime ), USE()<< p << pprime, USE()<< pprime << qprime, PROJECT()<< pprime );

// U2/3 (T X hatS)\(_ X BoTinv) // == {(p q)| exists p': (p T p') /\ (q in hS) /\ ~(p' BoTinv q)}
static composition_spec OP_UTXhSminusBoTinv(*this);
OP_UTXhSminusBoTinv.initialize3( (bool2&bool1&~bool0), ( p % pprime % q ), USE()<< p << pprime, USE()<< q, USE()<< pprime << q, PROJECT()<< pprime );

// U2/3 (hatS X Tinv)\(ToB X _) // == {(p q)| exists q': (p in hatS) /\ (q' Tinv q) /\ ~(p ToB q')}
static composition_spec OP_hSXTinvminusToB(*this);
OP_hSXTinvminusToB.initialize3( (bool2&bool1&~bool0), ( p % qprime % q ), USE()<< p, USE()<< qprime << q, USE()<< p << qprime, PROJECT()<< qprime );

//convenience:
int L2 = L+L;
int L3 = L+L+L;

MDDL::mdd B = Bin;
MDDL::mdd Bold = B;

// first, saturate lower levels:
B = BiSat1Saturationint( L, hatS, T2, T2inv, NQ2, B, cache );

// then do fixed point calculation:
do {
    Bold = B;
    // operate at this level:
    // apply each transition relation at this level (T is sorted by descending Top level):
    for ( int a=0; (a<NQ) && (T[a].level() == SXSL); ++a ) {

        MDDL::mdd Ta = T[a];
        MDDL::mdd Tinva = Tinv[a];

        // B o Tinv // == {(p' q)| exists q': p' B q' /\ q' Tinv q}
        MDDL::mdd BoTinv = OP_BoTinv.execute_on4( L3, L2, NULL, NULL, B, Tinva, MDD_GCOMPOSE43_QQ );

        // T o B // == {(p q')| exists p': p T p' /\ p' B q'}
        MDDL::mdd ToB = OP_ToB.execute_on4( L3, L2, NULL, NULL, Ta, B, MDD_GCOMPOSE44_QQ );

        // U2/3 (T X hatS)\(_ X BoTinv) // == {(p q)| exists p': (p T p') /\ (q in hS) /\ ~(p' BoTinv q)}
        MDDL::mdd UTXhSminusBoTinv = OP_UTXhSminusBoTinv.execute_on4( L3, L2, NULL, Ta, hatS, BoTinv, MDD_GCOMPOSE45_QQ );

        // U2/3 (hatS X Tinv)\(ToB X _) // == {(p q)| exists q': (p in hatS) /\ (q' Tinv q) /\ ~(p ToB q')}
        MDDL::mdd hSXTinvminusToB = OP_hSXTinvminusToB.execute_on4( L3, L2, NULL, hatS, Tinva, ToB, MDD_GCOMPOSE46_QQ );

        MDDL::mdd DeltaBbar = MDDL::g_mddf.or_qq(UTXhSminusBoTinv,hSXTinvminusToB);

        // put the results into B

        B = MDDL::g_mddf.minus_qq( B, DeltaBbar );

        // saturate those results:
        { if(DeltaBbar) if(B) B = BiSat1Saturationint( L, hatS, T2, T2inv, NQ2, B, cache ); };
    };
} while ( Bold != B );

MDDL::g_mddf.cache_add( cache, Bin, B );  // memoize

return B;
```

In the improved code, there is still a block of code near the beginning for initializing vectors, but here the code is less amorphous, and the vectors are hidden within objects of type composition_spec. Each composition_spec supplies all the information (except for parameters, sizes, and caching information) needed by a composition operation used later. This makes both the vector construction code and the composition operator code more readable. There is still considerable distance between the code and the algorithm pseudocode, as the set-theoretic notation of the pseudocode are not directly supported in C++. Additionally, cumbersome characteristics of the existing library interface are obvious, such as the use of function names (MDDL::g_mddf.minus_qq) that depend on which kind of MDDs are being used.

C++ does provide template meta-programming, which can be used to generate complex efficient code at compile time, while allowing slight coding syntax improvement. I propose to implement a library and interface that generates efficient code from input code closer to pseudocode algorithms for MDD operations. With the proposed improvements, I hope to instead write (something like) the following code:

typedef TeDDy::tupleset stateset;

typedef TeDDy::interleaved<stateset,2> intstaterelation;

Parameters: int L, stateset hatS, intstaterelation * T, intstaterelation * Tinv, int NQ, intstaterelation Bin, TeDDy::CacheGroup caches

```
// calculate T2, T2inv, and NQ2, for lower levels
stateset * T2 = T;
stateset * T2inv = Tinv;
int NQ2 = NQ;
const int hatSL = hatS.level();
const int TL1
const int SXSL = hatSL+hatSL;


while( NQ2 && (T2[0].level()==SXSL)){ T2++; T2inv++; NQ2--; };

// variables for evaluating B\((U2/3 (T X hatS)\(_ X BoTinv)) U (U2/3 (hatS X Tinv)\(ToB X _)))
// == B\(  {(p q)| exists p': (p T p') /\ (q in hatS) /\ ~(exists q': p' B q' /\ q' Tinv q)}
//       U {(p q)| exists q': (p in hatS) /\ (q' Tinv q) /\ ~(XXXexists p': p B p' /\ p' Tinv q'XXX)})
#TeDDy_4_Symbolic_Tuple_Names( p, p_prime, q, q_prime )

//convenience:
int L2 = L+L;
int L3 = L+L+L;

TeDDy::mdd B = Bin;
TeDDy::mdd Bold = B;

// first, saturate lower levels:
B = BiSat1Saturationint( L, hatS, T2, T2inv, NQ2, B, caches );

// then do fixed point calculation:
do {
    Bold = B;
    // operate at this level:
    // apply each transition relation at this level (T is sorted by descending Top level):
    for ( int a=0; (a<NQ) && (T[a].level() == SXSL); ++a ) {

        TeDDy::mdd Ta = T[a];
        TeDDy::mdd Tinva = Tinv[a];

        // B o Tinv // == {(p' q)| exists q': p' B q' /\ q' Tinv q}
        TeDDy::mdd BoTinv = tuples2<intstaterelation>(p_prime,q)(exists(q_prime)( B(p_prime,q_prime) & Tinva(q_prime,q) ));
        BoTinv.withcache(caches).calculate();

        // T o B // == {(p q')| exists p': p T p' /\ p' B q'}
        TeDDy::mdd ToB = tuples2<intstaterelation>(p,q_prime)(exists(p_prime)(Ta(p,p_prime) & B(p_prime,q_prime)));
        ToB.withcache(caches).calculate();

        // U2/3 (T X hatS)\(_ X BoTinv) // == {(p q)| exists p': (p T p') /\ (q in hS) /\ ~(p' BoTinv q)}
        TeDDy::mdd UTXhSminusBoTinv = tuples2<intstaterelation>(p,q)(exists(p_prime)(Ta(p,p_prime) & hatS(q) & ~BoTinv(p_prime,q)));
        UTXhSminusBoTinv.withcache(caches).calculate();

        // U2/3 (hatS X Tinv)\(ToB X _) // == {(p q)| exists q': (p in hatS) /\ (q' Tinv q) /\ ~(p ToB q')}
        TeDDy::mdd hSXTinvminusToB = tuples2<intstaterelation>(p,q)(exists(q_prime)(hatS(p) & Tinva(q_prime,q) & ~ToB(p,q_prime)));
        hSXTinvminusToB.withcache(caches).calculate();

        TeDDy::mdd DeltaBbar = UTXhSminusBoTinv .U hSXTinvminusToB;

        // put the results into B

        B = ( B - DeltaBbar ).withcache(caches).calculate();

        // saturate those results:
        { if(DeltaBbar) if(B) B = BiSat1Saturationint( L, hatS, T2, T2inv, NQ2, B, caches ); };
    };
} while ( Bold != B );

caches.add( &BiSatSaturationint, Bin, B );  // memoize

return B;
```

Note first that this code should not be taken as a specific definitive example, and that some library syntax and design issues remain open for resolution in the research phase of this project. In this code, there is no longer a separate block of code for initializing the control vectors for another block of code. Instead, the main block of code invokes a customized set of recursive functions which implicitly hold the control knowledge previously stored in the control vectors. Additional advantages of this improved coding style are obvious, including the following:

1. The use of types to indicate various forms of encoding, such as interleaved pairs vs. concatenated pairs.
2. The use of a single name to denote a given operator, independent of which type of tuple sets and reductions are used in the encoding.
3. The simple combination of multiple operators into efficient aggregate operations.
4. The use of comprehension-like structures, having local variable names, to define relations.

3.3 Non-algorithm library contributions

The above-mentioned algorithm library contributions enable the study of novel model checking algorithms and techniques described in this section.

3.3.1 Locality enhancement for LTS for weak bisimulation

My saturation-based bisimulation algorithm in Section 2.4.2 is weak in the cases where there are transition relations having large support. I expect that in many of these cases, such a transition relation may have useful projections onto *uncertain transition relation*s with relatively small support. An uncertain transition relation is a transition relation where some pairs in the relation have an imprecise specification for either (or both) the domain element or (or and) the range element. Such a projection is not necessarily lossless. I hope to automatically decompose transition relations with large support into multiple uncertain transition relations having small support. For the purpose of calculating $\sim$, both the original transition relations and the decomposed uncertain transition relations would be used. In the process of calculating $\sim$, the saturation based algorithm would give priority to use of the decomposed uncertain transition relations (as they have small support, hence expected lower usage cost) over the use of the original transition relations having large support. The hope is that, as with many other saturation-based algorithms, most of the useful work would be performed by usage of the transitions relations having small support, leaving less need for using transitions relations having large support, so that the overall run-time would decrease, compared with the previous algorithm.

3.3.2 Fully symbolic lumping algorithm

I expect to be able to manipulate the definition of lumping, given in Section 2.4.3 into a form directly applicable to a saturation-based solution analogously to how the definition of bisimulation was manipulated in Section 2.4.2. If I succeed in this endeavor, the resulting algorithm would be (nearly) the first fully symbolic lumping algorithm. The algorithm in [16](§7.49) is fully symbolic, but in a way that is somewhat arbitrary and unnatural. It is likely that such an algorithm would be the only feasible way to perform lumping on systems where there are very many equivalence classes. A full study of fully symbolic lumping requires the use of edge-valued GDDs (Section 4.1), which may or may not be available during the course of this research.

3.3.3 Re-organizing parallel saturation

The efficiency of saturation-based state-space exploration, and many other DD algorithms, is strongly influenced by the necessary ordering of variables, and by the support of related transition relations, as abbreviated in the form of event spans. The *event span* of a transition relation (given a specific variable ordering) is the smallest set of contiguous levels containing the support variables of the relation. For purposes of saturation-based state-space exploration, all other things being equal, a variable ordering which causes transition relations to have small event spans is preferred over a variable ordering which causes transition relations to have large event spans. Typically, before state-space exploration begins, the variables have been pre-ordered in such a way as to bring about reasonably small event spans when possible. This heuristic proposes to take further advantage of this ordering to identify opportunities for parallel firing of events with non-overlapping spans, during state-space exploration. Due to time limitations, I will simplify this description to the case where only two processors are available. Sequential saturation-based state-space exploration consists of: initializing a DD S with an encoding of the set of initial system states, followed by augmentation of S through firing individual events (transition relations), in the order indicated by the saturation heuristic, after which S contains an encoding of the entire reachable state-space. The saturation heuristic gives priority to events with a lower *top*, where the top of an event is the highest level of its event span. This heuristic divides the levels into two groups (high and low) and events into three groups (high, low, and hybrid). Each group (high and low) of levels is contiguous and comprises about half of the groups, where the exact point of division between them may be subject to tuning. The high group of events comprises events where the span is entirely within the high levels. Analogously, the low group of events comprises events where the span is entirely within the low levels. The remaining events comprise the hybrid group, which is likely to be small given a good variable ordering. However, any system with no hybrid events could simply be factored into multiple independent systems, hence realistic systems will always have at least one hybrid event. This technique alters the saturation order by making the high and low group of events independent of each other, while the high and low groups of events both have priority over the hybrid events. Within the DD encoding S, the nodes corresponding to high levels are separated from the nodes corresponding to the low levels by an extra invisible level of symbolic encoding, which allows concurrent firing of high events and low events, each operation only on the corresponding levels of S. When high and low events are no longer able to fire due to (possibly temporary) convergence of S, The extra level of symbolic encoding is processed to unify the higher and lower levels of S into a single DD, after which hybrid events may fire. When it is time to attempt firings of high and/or low events, the invisible level of symbolic encoding is re-imposed prior to such firings. The advantage of this scheme is that high events may be fired in parallel with low events, and there should be very few hybrid events. The disadvantages are that the invisible level of symbolic encoding must be processed, possibly many times, before proceeding, and that hybrid events must proceed sequentially. These disadvantages have not been quantified, yet their extent may (or may not) dominate the benefits of this technique.

4 Potential work

The following investigations may be performed as time allows. It is likely that most of these items are, in fact, future work.

4.1 Edge-valued GDDs

It is somewhat obvious that an edge-valued variation of GDDs (EVGDDs?), having the benefits of both EVMDDs and GDDs, is possible, and that a library for manipulating such data structures is a logical extension of the current proposal. However, as it may take some time to work out the theory of EVGDDs, it is not clear wether such an investigation can be completed within the time frame required of the current proposal.

4.2 Novel speculation heuristics for parallel GDD library

This research presents another opportunity to attempt to obtain the parallel scalability gains long hoped for by model checking researchers. The following techniques have not yet been explored, yet appear to present obvious opportunities for performance enhancement through parallelism. Rapid progress in this area is less likely due to my lack of familiarity with any locally available massively parallel computing platforms. Note that these descriptions below are merely summaries of the most important parts, made necessarily brief due to the impending deadlines, and in no way illustrate the full extent of my thinking on this subject. These three speculative techniques relate to library-level parallelism and could become performance improvements within the TeDDy library. Fundamental library operations on DDs involve traversals of homologous parts of each DD input involved in the operation. Taking the union, for example, of A and B involves coordinated traversal of homologous parts of A and B, invoking the union operation on homologous children (and further descendants) of A and B. A primary problem with the 'distribution by level' parallelism scheme described in Section 2.5.1 is that traversal requests sent from the processor holding a parent node to the processor holding the child nodes of that parent all arrive together, presenting plenty of work to the processor holding the child nodes, but not to any other processors, limiting the spread of parallel activity. These first three techniques speculatively initiate parallel activity at lower levels.

4.2.1 Forward speculation

As can be understood from the $union()$ algorithm in Section 2.1.2, the choice of which descendants of A and B are to be combined in a nested $union()$ operations cannot be reliably determined a priori. *Forward speculation* chooses some remote descendants of A and B based on information cached with A and B, possibly including a list of 'preferred' descendants, and always at a specific level. Thus, the $union()$ function applied to A and B will speculatively invoke $union()$ function applied to some pairs of various remote descendants of A with various remote descendants of B, potentially producing results which will be used in the construction of $union(A, B)$. In the case where the level of chosen descendants actually has very few descendants of A and/or B, this method is more likely to produce useful speedup.

4.2.2 Reverse speculation

Without speculation, (at lower levels of DD) $union(A, B)$ is calculated only if it is a necessary part of a higher-level $union()$ operation where A and B are homologous components of operands in the higher-level $union()$ operation. *Reverse speculation* speculatively calculates $union(A, B)$ based on information cached with A and B, possibly including information about the sequence of labels on paths which lead to A and to B. Such information might be stored in compressed form as a signature. Thus, when a higher-level $union()$ is requested, a speculative calculation of $union(A, B)$ may be launched if a signature of a path leading to A matches a signature of a path leading to B. This technique seems more likely to produce a useful speedup in cases where there are few paths leading to A and/or to B, allowing for efficient search for matching path signatures.

4.2.3 Count-based speculation

Count-based speculation considers all levels of the operands of a highest-level DD operation, to find levels where the upper bound on the number of output nodes is lowest. If the bound on the number of output nodes at a given level is below some carefully tuned threshold, speculative calculation of possible output nodes at that level is initiated. Aside from the cost of extra bookkeeping, this technique costs very little when the threshold is very low, although it may also be less likely to produce useful speedup.

5 Future work

These items are promising avenues of research which are likely to produce useful results given the expected success of the currently proposed research.

5.1 Additional encoding to promote sharing

It is possible, to potentially increase sharing by using an additional layer of encoding for variable names (references to members of the tuple argument of an encoded function). This additional layer of encoding may be thought of as renamings of variables occurring in a sub-tree, applied whenever the subtree is accessed by a specific edge, which will have an annotation giving the renaming. It seams that relatively few situations occur where sharing will be improved by such an additional layer of encoding, however, this has not been quantified, and so could be a subject of future research.

5.2 Real decision diagrams

As with edge-valued GDDs, it appears that a natural progression would be to extend the domain of GDDs to tuples of reals ($\mathbb{R}^*$) in addition to tuples of naturals. Dario D'amico has already explored extending the domain of MDDs to real tuples in his thesis on using *Real Decision Diagrams* (RDD's) [**?**](§**??**).

6 Schedule

Here I first enumerate the tasks involved in this research, and then propose a specific schedule for their execution.

6.1 Tasks

I propose the following plans for technical tasks be performed prior to dissertation, in addition to solving any further research problems that arise within these tasks. This list is somewhat foreshortened due to the 10-year time limit imposed on my scholastic duration by the Graduate Division (I must successfully defend my dissertation during or before 3Q2015, as I started in 4Q2005). Additionally, I prefer to defer any writing tasks (such as papers and other correspondence) until after the technical tasks are finished and the dissertation is started, so that enough research may be accomplished to support the dissertation.

1. Preparation of GDD-based model checking research platform.

   I will choose a suitable computation platform, which must be a roughly-symmetric parallel processor that offers scaling to at least 12 cores, with a current implementation of at least 4 cores, in a system available to me. An algorithm library, having the functionality of GDD operations, will be constructed on the chosen computation platform, with a well defined (but possibly overly verbose) API. This must be done keeping in mind the desired properties of the user library interface.

   Basic functionality shall be:

   (a) Construction of GDD-encoded characteristic functions representing Bundles.

   (b) GDD set operations with 0-6 GDD parameters with single quantification and result caching.

   (c) Checking of match between interleaved/non-interleaved parameter types and corresponding number of levels.

   (d) Automatic reduction choice (Section 3.1.7)

   Potential additional functionality may include:

   (e) Edge-Valued diagrams with sum quantification for EV+GDDs and sum and product quantification for EV*GDDs

   (f) Compression of node contents for common patterns and/or small indices

2. Construction of novel library interface.

   One or more layers of API will be added to the GDD library, to allow user coding of relatively elegant yet efficient model checking code. Assuming the language for the project is C++, this task will primarily involve template metaprogramming as described in Section 3.2.

3. Demonstration of improved library.

   I will adapt and analyze the bisimulation algorithms from our paper [38](§7.3), including my weak bisimulation algorithm (Section 2.4.2) and Wimmer's bisimulation algorithm [54](§7.54) using the new GDD library.

4. Locality enhancement for LTS for weak bisimulation.

   I will explore the use of locality enhancement to fully symbolic bisimulation, as described in Section 3.3.1, using the new GDD library.

5. Parallel Saturation Algorithm based on model locality.

   Based on the above parallel library implementation, I will measure the parallel speed-up of parallel saturation-based state-space exploration using the re-organized saturation scheme described in Section 3.3.3.

6. Fully Symbolic Lumping (potential task contingent on schedule).

   I will study the performance of a novel fully symbolic lumping algorithm, following the plan described in Section 3.3.2, using the new GDD library.

7. Practical Application (potential task contingent on schedule).

   A relevant model-checking application will be chosen and implemented using the new parallel GDD library and parallel saturation-based state-space exploration.

As opportunities arise, it will be advantageous to also perform some of the following publishing-related activities:

PA: Paper on the properties of GDDs.

PB: Report on the TeDDy with GDDs and the novel interface.

PC: Paper on Artificial Locality Enhancement, or Using Uncertainty to Improve Locality.

PD: Paper on Saturation-based Fully Symbolic Lumping.

PE: Papers on Parallel TeDDy implementation, and related parallel library algorithms.

PF: Papers on Model-Locality based parallel Saturation implementation.

PG: Papers on any additional research problems solved.

Some of the following additional activities will be necessary:

H: Writing and defending this proposal.

J: Apply for funding from NSF, or other sources.

K: Write Dissertation.

L: Defend Dissertation.

Task	2014			2015				
	Q2	Q3	Q4	Q1	Q2	Q3	Q4	
1		. . *	* * *					Library (multicore)
2		. . *	* * *					Interface
3				*				Bisimulation
4				*				Locality
5				*				Saturation
6					* *			Lumping
7					*			Application
H	. . . *	* *						Proposal
J		* * *						Seek Funding (oops)
K				* * *	* * *	* *		Write Dissertation
L						*		Defend Dissertation
P*						*	* . . .	Write Papers

Fig. 15: Schedule

7 Annotated Bibliography

I considered many papers from many conference proceedings, and many journals. After reading the abstracts (and frequently much more) of the many chosen papers, the approximately 55 papers and other publications discussed in this section were selected as being possibly relevant to the current proposal.

The publications fall into several categories:

1. Publications that relate to the theory of decision diagrams, model checking and related algorithms.

2. Publications that illustrate methods that provide scalable parallelism for some applications, that could hint at techniques that could be useful for parallel implementation of TeDDy.

3. Finally, publications that must be included in such proposals, although practically irrelevant.

I have attempted to list the publications in order corresponding to the above list of categories, although some publications fall into more than one of these categories. Each entry references the bibliography at the end of this document for full bibliographic information. Each entry also includes a link, usually to the relevant DOI page, so the reader may easily access the original works. In some cases there is also a link to an on-line copy of the publication itself. In one notable case [49](§7.32), there is a link to the on-line video of the presentation.

7.1 A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis [10]

Due to time constraints, I merely copy the abstract here:

> Chaining can reduce the number of iterations required for symbolic state-space generation and model-checking, especially in Petri nets and similar asynchronous systems, but requires considerable insight and is limited to a static ordering of the events in the high-level model. We introduce a two-step approach that is instead fine-grained and dynamically applied to the

decision diagrams nodes. The first step, based on a precedence relation, is guaranteed to improve convergence, while the second one, based on a notion of node fullness, is heuristic. We apply our approach to traditional breadth-first and saturation state-space generation, and show that it is effective in both cases.

DOI: http://dx.doi.org/10.1007/11901914_7

7.2 A Fully Symbolic Bisimulation Algorithm [37]

Due to time constraints, I merely copy the abstract here:

We apply the saturation heuristic to the bisimulation problem for deterministic discrete-event models, obtaining the fastest to date symbolic bisimulation algorithm, able to deal with large quotient spaces. We compare performance of our algorithm with that of Wimmer et al., on a collection of models. As the number of equivalence classes increases, our algorithm tends to have improved time and space consumption compared with the algorithm of Wimmer et al., while, for some models with fixed numbers of state variables, our algorithm merely produced a moderate extension of the number of classes that could be processed before succumbing to state-space explosion. We conclude that it may be possible to solve the bisimulation problem for systems having only visible deterministic transitions (e.g., Petri nets where each transition has a distinct label) even if the quotient space is large (e.g., 10^9 classes), as long as there is strong event locality.

DOI: http://dx.doi.org/10.1007/978-3-642-24288-5_19

7.3 AN EFFICIENT FULLY SYMBOLIC BISIMULATION ALGORITHM FOR NON-DETERMINISTIC SYSTEMS [38]

Due to time constraints, I merely copy the abstract here:

The definition of bisimulation suggests a partition-refinement step, which we show to be suitable for a saturation-based implementation. We compare our fully symbolic saturation-based implementation with the fastest extant bisimulation algorithms over a set of benchmarks, and conclude that it appears to be the fastest algorithm capable of computing the largest bisimulation over very large quotient spaces.

DOI: http://dx.doi.org/10.1142/S012905411340011X

7.4 Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits [28]

Due to time constraints, I merely copy the abstract here:

This paper presents a scalable method for parallel symbolic reachability analysis on a distributed-memory environment of workstations. our method makes use of an adaptive partitioning algorithm which achieves high reduction of space requirements. The memory balance is maintained by dynamically repartitioning the state space throughout the computation. A compact BDD representation allows coordination by shipping BDDs from one machine to another, where different variable orders are allowed. The algorithm uses a distributed termination

protocol which none of the memory modules preserving a complete image of the set of reachable states. No external storage is used not the disk; rather, we make use of the network which is much faster. We implemented our method on a standard, loosely-connected environment of workstations, using a high-performance model checker. Our initial performance evaluation using several large circuits shows that our method can handle models that are too large to fit in the memory of a single node. The efficiency of the partitioning algorithm is linear in the number of workstations employed, with a 40-60% efficiency. A corresponding decrease of space requirements is measured throughout the reachability analysis. Our results show that the relatively-slow network does not become a bottleneck, and that computation time is kept reasonably small.

link: http://www.cs.technion.ac.il/users/orna/CAV00-scalable.ps

7.5 A Work-Efficient Distributed Algorithm for Reachability Analysis [26]

Due to time constraints, I merely copy the abstract here:

This work presents a novel distributed, symbolic algorithm for reachability analysis that can effectively exploit, as needed, a large number of machines working in parallel. The novelty of the algorithm is in its dynamic allocation and reallocation of processes to tasks and in its mechanism for recovery, from local state explosion. As a result, the algorithm is work-efficient: it utilizes only those resources that are actually needed. In addition, its high adaptability makes it suitable for exploiting the resources of very large and heterogeneous distributed, non-dedicated environments. Thus, it has the potential of verifying very large systems. We implemented our algorithm in a tool called Division. Our preliminary experimental results show that the algorithm is indeed work-efficient. Although that the goal of this research is to check larger models, the results also indicate the potential to obtain high speedups, because communication overhead is very small.

DOI: http://dx.doi.org/10.1007/978-3-540-45069-6_5

7.6 Scalable Distributed On-The-Fly Symbolic Model Checking [3]

Due to time constraints, I merely copy the abstract here:

This paper presents a scalable method for parallel symbolic on-the- fly model checking on a distributed-memory environment of workstations. Our method combines a parallel version of an on-the-fly model checker for safety properties with a scalable scheme for reachability analysis. The extra load of storage required for counter example generation is evenly distributed among the processes by our memory balancing. For the sake of scalability, at no point during computation the memory of a single process contains all the data from any of the cycles. The counter example generation is thus performed through collaboration of the parallel processes. We develop a method for the counter example generation keeping a low peak memory requirement during the backward step and the computation of the inverse transition relation. We implemented our method on a standard, loosely-connected environment of workstations, using a high-performance SMV-based model checker. Our initial performance evaluation using several large circuits shows that our method can check models that are too large to fit in the memory of a single node. Our on-the-fly approach may find counter examples even when the model is too large to fit in the memory of the parallel system.

DOI: http://dx.doi.org/10.1007/3-540-40922-X_24

7.7 Achieving Speedups in Distributed Symbolic Reachability Analysis through Asynchronous Computation [25]

Due to time constraints, I merely copy the abstract here:

> This paper presents a novel BDD-based distributed algorithm for reachability analysis which is completely asynchronous. Previous BDD-based distributed schemes are synchronous: they consist of interleaved rounds of computation and communication, in which the fastest machine (or one which is lightly loaded) must wait for the slowest one at the end of each round. We make two major contributions. First, the algorithm performs image computation and message transfer concurrently, employing non-blocking protocols in several layers of the communication and the computation infrastructures. As a result, regardless of the scale and type of the underlying platform, the maximal amount of resources can be utilized efficiently. Second, the algorithm incorporates an adaptive mechanism which splits the workload, taking into account the availability of free computational power. In this way, the computation can progress more quickly because, when more CPUs are available to join the computation, less work is assigned to each of them. Less load implies additional important benefits, such as better locality of reference, less overhead in compaction activities (such as reorder), and faster and better workload splitting. We implemented the new approach by extending a symbolic model checker from Intel. The effectiveness of the resulting scheme is demonstrated on a number of large industrial designs as well as public benchmark circuits, all known to be hard for reachability analysis. Our results show that the asynchronous algorithm enables efficient utilization of higher levels of parallelism. High speedups are reported, up to an order of magnitude, for computing reachability for models with higher memory requirements than was previously possible.

DOI: http://dx.doi.org/10.1007/11560548_12

7.8 Verifying Very Large Industrial Circuits Using 100 Processes and Beyond [22]

Due to time constraints, I merely copy the abstract here:

> Recent advances in scheduling and networking have cleared the way for efficient exploitation of large-scale distributed computing platforms, such as computational grids and huge clusters. Such infrastructures hold great promise for the highly resource-demanding task of verifying and checking large models, given that model checkers would be designed with a high degree of scalability and flexibility in mind. In this paper we focus on the mechanisms required to execute a high-performance, distributed, symbolic model checker on top of a large-scale distributed environment. We develop a hybrid algorithm for slicing the state space and dynamically distribute the work among the worker processes. We show that the new approach is faster, more effective, and thus much more scalable than previous slicing algorithms. We then present a checkpoint-restart module that has very low overhead. This module can be used to combat failures which become probable with the size of the computing platform. However, checkpoint-restart is even more handy for the scheduling system: it can be used to avoid reserving large numbers of workers, thus making the distributed computation work-efficient. Finally, we discuss for the first time the effect of reorder on the distributed model checker and show how the distributed system performs more efficient reordering than the sequential one. We implemented our contributions on a network of 200 processors, using a distributed scalable scheme that employs a

high-performance industrial model checker from Intel. Our results show that the system was able to verify real-life models much larger than was previously possible.

DOI: http://dx.doi.org/10.1007/11562948_4

7.9 Roomy: A System for Space Limited Computations [31]

Due to time constraints, I merely copy the abstract here:

There are numerous examples of problems in symbolic algebra in which the required storage grows far beyond the limitations even of the distributed RAM of a cluster. Often this limitation determines how large a problem one can solve in practice. Roomy provides a minimally invasive system to modify the code for such a computation, in order to use the local disks of a cluster or a SAN as a transparent extension of RAM.

Roomy is implemented as a C/C++ library. It provides some simple data structures (arrays, unordered lists, and hash tables). Some typical programming constructs that one might employ in Roomy are: map, reduce, duplicate elimination, chain reduction, pair reduction, and breadth-first search. All aspects of parallelism and remote I/O are hidden within the Roomy library.

DOI: http://dx.doi.org/10.1145/1837210.1837216

7.10 Parallel Disk-Based Computation for Large, Monolithic Binary Decision Diagrams [32]

Due to time constraints, I merely copy the abstract here:

Binary Decision Diagrams (BDDs) are widely used in formal verification. They are also widely known for consuming large amounts of memory. For larger problems, a BDD computation will often start thrashing due to lack of memory within minutes. This work uses the parallel disks of a cluster or a SAN (storage area network) as an extension of RAM, in order to efficiently compute with BDDs that are orders of magnitude larger than what is available on a typical computer. The use of parallel disks overcomes the bandwidth problem of single disk methods, since the bandwidth of 50 disks is similar to the bandwidth of a single RAM sub-system. In order to overcome the latency issues of disk, the Roomy library is used for the sake of its latency-tolerant data structures. A breadth-first algorithm is implemented. A further advantage of the algorithm is that RAM usage can be very modest, since its largest use is as buffers for open files. The success of the method is demonstrated by solving the 16-queens problem, and by solving a more unusual problem — counting the number of tie games in a three-dimensional 4x4x4 tic-tac-toe board.

DOI: http://dx.doi.org/10.1145/1837210.1837222

7.11 Distributed Saturation [8]

Due to time constraints, I merely copy the abstract here:

The Saturation algorithm for symbolic state-space generation, has been a recent breakthrough in the exhaustive verification of complex systems, in particular globally-asynchronous/locally-synchronous systems. The algorithm uses a very compact Multiway Decision Diagram (MDD)

encoding for states and the fastest symbolic exploration algorithm to date. The distributed version of Saturation uses the overall memory available on a network of workstations (NOW) to efficiently spread the memory load during the highly irregular exploration. A crucial factor in limiting the memory consumption during the symbolic state-space generation is the ability to perform *garbage collection* to free up the memory occupied by dead nodes. However, garbage collection over a NOW requires a nontrivial communication overhead. In addition, operation cache policies become critical while analyzing large-scale systems using the symbolic approach. In this technical report, we develop a garbage collection scheme and several operation cache policies to help on solving extremely complex systems. Experiments show that our schemes improve the performance of the original distributed implementation, SmArTNow, in terms of time and memory efficiency.

`URL: http://ntrs.nasa.gov/search.jsp?R=20070017995`

7.12 Caching, Hashing, and Garbage Collection for Distributed State Space Construction [9]

Due to time constraints, I merely copy the abstract here:

> The Saturation algorithm for symbolic state-space generation is a recent advance in exhaustive verification of complex systems, in particular globally-asynchronous/ locally-synchronous systems. The distributed version of Saturation uses the overall memory available on a network of workstations (NOW) to efficiently spread the memory load during its highly irregular exploration. A crucial factor in limiting the memory consumption in symbolic state-space generation is the ability to perform garbage collection to free up the memory occupied by dead nodes. However, garbage collection over a NOW requires a nontrivial communication overhead. In addition, operation cache policies become critical while analyzing large-scale systems using a symbolic approach. In this paper, we develop a garbage collection scheme and several operation cache policies to help the analysis of complex systems. Experiments show that our schemes improve the performance of the original distributed implementation, SmartNOW, in terms of both time and memory efficiency.

`URL: http://www.cs.ucr.edu/~ciardo/pubs/2007PDMC-DistSMART.pdf`

7.13 Parallelising symbolic state-space generators [17]

Due to time constraints, I merely copy the abstract here:

> Symbolic state-space generators are notoriously hard to parallelise, largely due to the irregular nature of the task. Parallel languages such as Cilk, tailored to irregular problems, have been shown to offer efficient scheduling and load balancing. This paper explores whether Cilk can be used to efficiently parallelise a symbolic state-space generator on a shared-memory architecture. We parallelise the Saturation algorithm implemented in the SMART verification tool using Cilk, and compare it to a parallel implementation of the algorithm using a thread pool. Our experimental studies on a dual-processor, dual-core PC show that Cilk can improve the run-time efficiency of our parallel algorithm due to its load balancing and scheduling efficiency. We also demonstrate that this incurs a significant memory overhead due to Cilk's inability to support pipelining, and conclude by pointing to a possible future direction for parallel irregular languages to include pipelining.

DOI: http://dx.doi.org/10.1007/978-3-540-73368-3_31

7.14 Parallel symbolic state-space exploration is difficult, but what is the alternative? [13]

Due to time constraints, I merely copy the abstract here:

> State-space exploration is an essential step in many modeling and analysis problems. Its goal is to find the states reachable from the initial state of a discrete-state model described. The state space can used to answer important questions, e.g., "Is there a dead state?" and "Can N become negative?", or as a starting point for sophisticated investigations expressed in temporal logic. Unfortunately, the state space is often so large that ordinary explicit data structures and sequential algorithms cannot cope, prompting the exploration of (1) parallel approaches using multiple processors, from simple workstation networks to shared-memory supercomputers, to satisfy large memory and runtime requirements and (2) symbolic approaches using decision diagrams to encode the large structured sets and relations manipulated during state-space generation. Both approaches have merits and limitations. Parallel explicit state-space generation is challenging, but almost linear speedup can be achieved; however, the analysis is ultimately limited by the memory and processors available. Symbolic methods are a heuristic that can efficiently encode many, but not all, functions over a structured and exponentially large domain; here the pitfalls are subtler: their performance varies widely depending on the class of decision diagram chosen, the state variable order, and obscure algorithmic parameters. As symbolic approaches are often much more efficient than explicit ones for many practical models, we argue for the need to parallelize symbolic state-space generation algorithms, so that we can realize the advantage of both approaches. This is a challenging endeavor, as the most efficient symbolic algorithm, Saturation, is inherently sequential. We conclude by discussing challenges, efforts, and promising directions toward this goal.

DOI: http://dx.doi.org/10.4204/EPTCS.14.1

7.15 Implementation of an Efficient Parallel BDD Package [46]

Due to time constraints, I merely copy the abstract here:
DOI: http://dx.doi.org/10.1145/240518.240639

7.16 AN ANTICIPATED FIRING SATURATION ALGORITHM FOR SHARED-MEMORY ARCHITECTURES [20]

Due to time constraints, I merely copy the abstract here:

> Parallelising symbolic state-space generation algorithms, such as Saturation, is known to be difficult as it often incurs high parallel overheads. To improve efficiency, related work on a distributed-memory implementation of Saturation proposed using idle processors for speculatively firing events and caching the obtained results, in the hope that these results will be needed lateron. This paper investigates a variant of this anticipated firing approach for shared-memory architectures, such as multi-core PCs. Rather than parallelising Saturation, the idea is to run the sequential Saturation algorithm on one core, while the others are given speculative work. Since computing the optimal strategy for selecting useful work is likely to be

an NP-complete problem, the paper devises and implements various heuristics. The obtained experimental results show that moderate speed-ups can be achieved as a result of using anticipated firing. However, the proposed heuristics require further work in order to be truly useful in practice.

link: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.7255

7.17 To Parallelize or to Optimize? [19]

Due to time constraints, I merely copy the abstract here:

> Model checking is a popular and successful technique for verifying complex digital systems. Carrying this techniqueand its underlying state-space generation algorithmsbeyond its current limitations presents itself with a number of alternatives. Our focus is on parallelization which is made attractive by the current trend in hardware architectures towards multi-core, multi-processor systems. The main obstacle in this endeavour is that, in particular, symbolic state-space generation algorithms are notoriously hard to parallelize. In this article, we describe the process of taking a sequential symbolic state-space generation algorithm, namely a generic, symbolic BFS algorithm, through a sequence of optimizations that leads up to the Saturation algorithm and follow the impact these sequential algorithms have on their parallel counterparts. In particular, we develop a parallel version of Saturation, discuss the challenges faced in its design and conduct extensive experimental studies of its implementation. We employ rigorous analysis tools and techniques for measuring and evaluating parallel overheads and the quality of the parallelization. The outcome of these studies is that the performance of a parallel symbolic state-space generation algorithm is almost impossible to predict and highly dependent on the model to which it is applied. In most situations, perceivable speed-ups are hard to achieve, but real-world applications where our technique produces significant improvements do exist. Nevertheless, it appears that time is better invested in optimizing sequential symbolic model checking algorithms rather than parallelizing them.

DOI: http://dx.doi.org/10.1093/logcom/exp006

7.18 A PARALLEL SATURATION ALGORITHM ON SHARED MEMORY ARCHITECTURES [21]

Due to time constraints, I merely copy the abstract here:

> Symbolic state-space generators are notoriously hard to parallelize. However, the Saturation algorithm implemented in the SMART verification tool differs from other sequential symbolic state-space generators in that it exploits the locality of firing events in asynchronous system models. This paper explores whether event locality can be utilized to efficiently parallelize Saturation on shared-memory architectures. Conceptually, we propose to parallelize the firing of events within a decision diagram node, which is technically realized via a thread pool. We discuss the challenges involved in our parallel design and conduct experimental studies on its prototypical implementation. On a dual-processor dualcore PC, our studies show speed-ups for several example models, e.g., of up to 50 % for a Kanban model, when compared to running our algorithm only on a single core.

link: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.114.1039

7.19 Can Saturation be Parallelised? [18]

Due to time constraints, I merely copy the abstract here:

> Symbolic state-space generators are notoriously hard to parallelise. However, the Saturation algorithm implemented in the SMART verification tool differs from other sequential symbolic state-space generators in that it exploits the locality of firing events in asynchronous system models. This paper explores whether event locality can be utilised to efficiently parallelise Saturation on shared-memory architectures. Conceptually, we propose to parallelise the firing of events within a decision diagram node, which is technically realised via a thread pool. We discuss the challenges involved in our parallel design and conduct experimental studies on its prototypical implementation. On a dual-processor dual-core PC, our studies show speed-ups for several example models, e.g., of up to 50% for a Kanban model, when compared to running our algorithm only on a single core.

`DOI: http://dx.doi.org/10.1007/978-3-540-70952-7_23`

7.20 The Fortress Language Specification [1]

The Fortress programming language aims to provide ways to do multithreaded and parallel programming while still retaining some of the look of traditional programming languages and accounting for the lessons learned from the last few decades. Fortress provides a relatively traditional data type system, perhaps describable as a cross between Java types and functional types, allowing inheritance between interfaces (traits ala Java interfaces), but without traditional Class types. Most control structures in Fortress are parallel by default, with some allowing the keyword "sequential" to specify sequential behavior. Several syntactic improvements enable the use of relatively math-like notation where appropriate, and include additional opportunities for default parallelism. In particular, Set, Array, and Map comprehensions involve implicit multithreading. The syntactic enhancements also include additional mathematical operator syntax, but come at some cost, in that many rules are needed to understand exactly how some expressions will be parsed.

A hierarchical system of regions is provided to allow program control of placement of threads, and possibly data structures, although it is not clear how to control the distribution of an array. (Possibly nested) atomic statements are provided so the programmer can avoid race conditions involving shared variables.

Static parameters may be used with functions and Object types, similarly to template parameters in C++.

Contracts allow specification of preconditions, postconditions, and invariants. These must be executable, however.

Overall, ignoring certain enhancements, Fortress is about what I would expect from an effort to extend a 'normal' language, such as Java, to include support for multithreaded series-parallel computation for 'normal' processor systems, such as clusters of x86 multicores. It is an improvement over some previous attempts at parallel languages, and may remain applicable for some time. I can however easily see the possibility of a much better language being implemented as part of the current work.

Of particular interest to me is their specification of Symmetric Multiple Dispatch, the implementation of which is described in a separate paper. This relates to the repair of the Ephemeral language, which may require some sort of multiple dispatch. It is remotely possible that Fortress will make a good target language for an Ephemeral compiler.

link: http://labs.oracle.com/projects/plrg/fortress.pdf

7.21 The Scala Language Specification Version 2.9 [39]

Scala is a Java-like language with numerous extensions, supporting the development of "domain-specific" languages as libraries within Scala. To this end, parametric (with type parameters) and ad-hoc polymorphic object oriented programming is carefully supported in the language. For example, looping control structures are "virtualized", so that those control structures (say "for" loops) are syntactic sugar for something like iterator calls to the loop variable, so the meaning of loops can be changed by the type of the loop variable. Also, "Implicits" allow some call parameters to be automatically inserted by the compiler, increasing the power of user-defined libraries. An "implicit" parameter, when not given at a call site will be automatically be set to an "implicit" value of the appropriate type if exactly one is visible in the call scope. The actual rules for implicits are more complex, and also allow implicit values to be automatically used as type conversion functions, called "views". The standard library also has some predefined implicits, that allow a user-defined (polymorphic) library to obtain a "manifest" structure describing the type of data as seen by the library user.

There is considerable syntactic flexibility in definition of operator names, allowing library authors to enhance the appearance of code that uses their libraries. Scala also uses the Unicode character set, with its extra operator characters, and allows XML embedding.

The standard implementation compiles to JVM byte codes. That feature made me initially doubt the relevance of this language, since I don't consider the JVM to be a likely platform for massively parallel processing. In fact, there is no mention of parallelism or even threading in the language or the standard library specification, although all the Java libraries are fully accessible.

However, Scala seems to be near the cutting edge of research on hosting application-specific mini languages, which is an alternative approach to some goals of the current proposal. Scala library authors may effectively define (using OOP and meta-programming-like techniques) mini-languages with which to write application code.

Whereas I propose to allow user-defined code transformations in the optimization process for a declarative language, Scala allows users to embed (and optimize in any manner they wish) application-specific languages, allowing the library author and user to guarantee good performance.

The primary weakness I see with this approach, is that Scala does not enforce correctness of such libraries, as they are user-defined and specified, and the language does not provide for their formal specification. Thus, a functional error in the object code could be due to either the user or to the library author. The current proposal requires that code transformations refine entailment, so that functional errors in object code are always traceable to the application specification.

In the arena of dynamically typed languages, it is useful to remember that lisp has long been used to host embedded mini-languages, through the use of the macro mechanism.

link: http://www.scala-lang.org/node/198

7.22 Scala-Virtualized [35]

Scala-Virtualized is an extension, of the Scala language, that provides improved facilities for hosting domain-specific languages. The three major extensions are as follows: 1. "infix methods" improve syntactic flexibility by effectively allowing the addition of class members externally to the definition of the class. 2. "Fully virtualized" program structures. All program constructs are considered syntactic sugared versions of method calls, rather than just looping constructs as with Scala originally. Consequently, the object system can convert what appears to be a plain Scala program into a strongly typed abstract syntax tree which can then be processed by a user-defined library to produce transformed code. 3. Additional information about source code files is made available to library code, so that user-defined libraries may produce more meaningful error messages when used incorrectly.

Scala-Virtualized has been successfully used to embed SQL queries within Scala programs in a type-safe manner, among other impressive achievements. As Scala-Virtualized is being developed by the developers of Scala, I imagine this represents the next step in the evolution of Scala. Likewise, my comments on the Scala language also apply to Scala-Virtualized. There is no strong support for formal proof of correctness, and no strong support for parallelism or even multi-threading apart from the Java libraries.

DOI: http://dx.doi.org/10.1145/2103746.2103769

7.23 Leveraging Data-Structure Semantics for Efficient Algorithmic Parallelism [15]

This paper describes a novel programmer-assisted method for parallelizing code, based on the understanding that efficient methods for determining some program properties necessary for efficient parallelism cannot be anticipated by the compiler writer. In particular, the data footprint of a computation or of a sub-computation cannot always be represented efficiently as list of memory locations. Also, the footprint cannot always be determined statically, due to the data-dependent nature of many computations, as well as the use of pointers and indexing. The data footprint of computations can be useful for determining which computations (from existing sequentially written code) can be run in parallel. In particular, operations with non-overlapping memory footprint may be executed in parallel. Unfortunately, comparing footprints based on lists of memory locations is infeasible for parallelizing significant programs.

The described system provides C++ templates and run-time components that allow a programmer to provide additional information to enable effective parallelization. The system represents the memory footprint of an operation abstractly, using types supplied by the programmer. The programmer also provides notifications of changes in the memory footprint of an operation, as well as a way of checking overlap (and also checking probability of overlap) in footprints represented by the programmer-supplied types. At run-time, the system calls the programmer-supplied checking functions to determine suitability of allowing parallel execution of the available operations. The system does this in 2 ways. In systems with software transactional memory (STM), this information is used to throttle concurrency based on probability of rollback. For other systems, it is used to obtain guarantees of non-interference before allowing parallel execution.

This can work well, because the programmer often knows an efficient representation of the desired footprint. For example, operations that each perform incremental operations on trees, then recursively descending on subtrees, have a footprint that can be conservatively be represented by reference to the node on which they are currently operating, the footprint being the subtree under that node. In this case, overlap between two footprints can be checked by inspecting the relation between the two referenced nodes.

The disadvantage of this approach (aside from the obvious fact that this implementation only applies to C++) is that there are no correctness guarantees as there are with some systems that rely on static analysis.

This is a very good example of a system that utilizes programmer knowledge (expressed procedurally, in this case) that potentially goes far beyond what can be anticipated by the toolmaker (or compiler writer), to enable greater program performance improvements.

DOI: http://dx.doi.org/10.1145/2016604.2016638

7.24 Distillation with Labelled Transition Systems [27]

The "Distillation" transform for functional programs was introduced in 2007 as an automatic way to perform certain transforms of list processing code previously thought to require mathematical insight. This paper discusses an explanation of how the Distillation transform sometimes produces a genuine algorithmic improvement having super-linear speedup. A correctness proof is also sketched.

This is vaguely reminiscent of a 1984 Lisp conference paper subtitled “Listlessness is better than laziness”.

This paper provides yet another example of the many optimization techniques that are somewhat specialized and hence not desirable to incorporate into a general purpose compiler, yet are highly desirable for many programs, necessitating its use in some compilers.

The proposed work solves this dilemma by providing a framework in which a user may safely define and invoke this optimization without modifying the compilation system itself.

DOI: http://dx.doi.org/10.1145/2103746.2103753

7.25 StagedSAC: A Case Study in Performance-Oriented DSL Development. [50]

This paper describes two implementations of SAC (Single Assignment C)-like languages in the Scala-Virtualized framework. SAC is a C-like language with a single-assignment rule for variables, and some additional array manipulation constructs intended for parallel implementation. In the first implementation, “LibrarySAC”, a library is defined, utilizing the syntactic flexibility of Scala to provide the programmer with a way of writing SAC-like code within Scala. SAC functionality is then available within Scala through the use of library calls.

The second implementation, “StagedSAC”, is accessed through library calls within Scala-Virtualized, but achieves higher performance through code optimizations and a “lightweight modular staging” (LMS) framework, also implemented in Scala-Virtualized. The full virtualization of Scala-Virtualized allows the library code to extract abstract syntax trees of the StagedSAC portion of the code. The library, at compile time, then performs transformations and optimizations on the code, resulting in improved Scala code. A constraint system on array shapes is derived from the program graph, the solution of which sometimes provides partial information on array shapes. Whatever partial information was derivable at compile time about array shapes is then used to optimize array code, via such improvements as removal of redundant index bounds checks, and later, loop specialization. All this information is passed to the LMS framework, which provides common optimizations such as constant folding, CSE, code motion, etc. Other optimizations such as tiling may be applied to improve cache effectiveness. Upon request, the Delite framework may also be used to do additional optimizations and generation of GPU code. One of the major contributions of this work is the ability to use <u>partial</u> shape information in certain ways for optimizing array code.

A results section provides comparisons between various levels of optimization of the StagedSAC implementation (executing on JVM) and a native SAC implementation, for some sample array programs (not using GPU). The fully implemented StagedSAC programs running on JVM had run-times within an order of magnitude of that of their corresponding natively compiled SAC programs.

This illustrates Scala being used for what it does best, the implementation and customized optimization of “Domain-Specific” languages within the bounds of interoperability within a general purpose strongly typed language.

DOI: http://dx.doi.org/10.1145/2103746.2103762

7.26 Active Pebbles: Parallel Programming for Data-Driven Applications [53]

Active Pebbles attempts to provide a framework for message-passing parallel programs that naturally utilize small messages with little or no coherence or predictability in the message flow patterns. Such programs may experience performance problems on traditional platforms, due to their high per-message overhead for inter-processor communications. Pebbles are (potentially very small) messages sent between entities. There may be very many senders, and very many receivers, called targets (also potentially very small). One may imagine the data flow pattern as a storm in a room full of tiny pebbles. There is no expectation of regular patterns or coherence. They provide results showing that Active Pebbles gives good performance for various benchmarks on top of various parallel programming platforms. The authors list 5 mechanisms responsible for the performance of their framework.

1 Fine-Grained Pebble Addressing: Pebbles are addressed directly to their target, with an address of reasonable size.

2 Message Coalescing: Pebbles may be aggregated temporarily into larger messages by the AP mechanisms, to decrease the messaging overhead on platforms that only efficiently support larger messages.

3 Active Routing: Pebble flow is slightly adjusted, to increase the probability of message coalescing.

4 Message Reduction: Depending on program semantics, certain pebbles may be reduced in transit, if they are coalesced into the same message.

5 Termination Detection: Support is provided for some distributed quiescence detection algorithms.

I find 1, 2, and 3 most interesting, as these are the mechanisms that overcome the small-message penalty on some platforms. This occurs with the penalty of small additional latencies introduced by the active routing and coalescence mechanisms. Fortunately, these mechanisms are programmable, and can be adjusted to also efficiently handle workloads with well-understood communication patterns. The observed performance justifies my assumption (in Ephemeral) that small messages (within the computing/programming model) are reasonably efficient and sufficient for all parallel programming needs.

This may provide a convenient implementation target for the parallel compiler, if there is not sufficient time to implement Ephemeral. If there is time to implement Ephemeral, it would be wise to consider use of some mechanisms from Active pebbles.

DOI: http://dx.doi.org/10.1145/1995896.1995934

7.27 Position paper: Using a "Codelet" Program Execution Model for Exascale Machines. [56]

The authors observe that course-grained parallelism, of the type that is efficiently supported by extant architectures, denies the run-time system of certain opportunities for adaptation, and tends to require large overheads for certain operations, such as task swapping and migration. The authors claim that their work indicates improvements are possible by dividing the program into smaller pieces (codelets) for execution.

The Codelet model they propose appears to be a hybrid between Ephemeral and the dataflow model.

It is difficult for me to see this work as original, as I have seen many hybrid dataflow systems proposed since the early 1980's, and the dataflow model is certainly no stranger to the use of small executable units. The part that seems novel to me is the claim that their work supports this. It's too bad the paper gives no details about, or references to such work. But I suppose that is to be expected from a paper that mentions IBM Cyclops-64, and Intel's single chip cloud machine, but not Sun's Niagara.

A generic hardware model is also mentioned, having a hierarchy of 4 levels (system, node, chip, and cluster), the typical unit for each level containing an interconnect and multiple units of a lower level attached to the interconnect. The node level unit also has extra DRAM banks attached to its interconnect. The lowest, cluster, level has computing units (CU) and at least one scheduling unit (SU), and a cluster memory, and interconnect between them. CUs and SUs have local memory and multiple register sets. SUs also have the ability to communicate with SUs in other clusters, presumably for load balancing and migration.

The relation between the hardware model and the codelet model is not clearly explained. My guess is that this paper was trimmed from a larger paper, and the hardware model was left in to provide some notion of what kind of computer would benefit from use of the codelet model.

DOI: http://dx.doi.org/10.1145/2000417.2000424

7.28 Adaptive Runtime Selection of Parallel Schedules in the Polytope Model [42]

This is limited to Polytope model computations, but has improved accuracy compared to many other techniques. The limitation to possibly parametric polytope model computations allows analytical determination of loop iteration counts and array sizes after the input array sizes are known. This, in turn, allows improved performance prediction.

The compiler may produce multiple versions of the code depending on how flexible the array data dependencies are. Their method produces a modified code at compile time which performs profiling on itself, along with the production code for performance execution. At install time, each code version is profiled systematically, with various numbers of threads, various input array sizes/shapes, and various tile sizes. The profile data, after tabulation, effectively defines a performance model for the code. Finally, at run time, the input sizes are known, and the number of available processors/threads is known approximately, so that the model can be used to predict the performance of the best variation of each version of the code. For each version, the best parameters can be predicted, such as number of threads to use, and tile sizes, depending on input sizing and processor loading, etc.

This method performs quite well at selecting a good version of the code to execute, because it is able to account for almost all factors influencing performance. These include: processor design (considered by load-time profiling), Input data size/shape (accounted for by systematic profiling using Polytope model) Processor resource availability/loading (via profiling with various numbers of threads)

What I find relevant about this work is that it explores one of many instances where a large class of computational problems (HPC codes) which are in general difficult to optimize, has a frequently occurring subset of problems (in this case, problems expressible as coherent loop nests with simple dependencies) which, as this research now shows, are relatively easy to optimize well, even for parallel processing.

My proposed project will provide a framework in which such cases may be defined and recognized, and in which to express what needs to be done in such cases, without requiring actual compiler modification.

link: http://dl.acm.org/citation.cfm?id=2048588

7.29 The Elephant and the Mice: The Role of Non-Strict Fine-Grained Synchronization for Modern Many-Core Architectures [43]

This paper explores 4 questions, among them is the performance gain achievable by the use of non-strict fine-grained synchronization (dataflow tokens, full/empty tag bits, synchronization state buffer), as compared with other synchronization mechanisms (barriers, signal-wait).

The authors implemented three fine-grained synchronization mechanisms for the (single-chip) IBM Cyclops-64, and tested it in very carefully crafted simulations to ensure accuracy of results. The IBM Cyclops-64 has a relatively symmetric NUMA architecture with a large crossbar and 160 “thread units” (single issue cores) in pairs having a shared FPU and a crossbar port. Each group of 10 thread units also share an instruction cache, every four of which also share a crossbar port. Each core has its own data memory, and direct (but higher latency) access to all other cores data memories through the crossbar. There are no data caches. The cores have scoreboarding which allows some out-of-order execution and write back. There is also a common high-speed synchronization bus shared by all cores.

The mechanism(s) implemented were integrated deeply into the architecture in the form of an Extended Synchronization State Buffer (ESSB). The ESSB essentially simulates the use of imaginary tag bits attached to some memory locations chosen implicitly by the program. Special load and store instructions utilize the ESSB. In the most effective mechanism tried, ESSB3, a special store sets the full bit associated with the memory location, after which the special load instruction resets it. If the load instruction occurs before the needed value has been stored to the location, a stall may eventually occur, but the thread will be automatically resumed once the data becomes available. The load instruction still issues but the thread will not stall until the value itself is needed by an operation.

The ESSB mechanisms, along with barriers and signal-wait, were all tested with customized versions of a number of benchmarks. As hoped for, the ESSB3 versions of all programs scaled well as long as there was available parallelism, while performance of the equivalent versions using other mechanisms became limited due to synchronization overhead. In particular, a case of the wavefront benchmark gave almost linear speed up out to 160 threads with ESSB3, while the best of the other methods (signal-wait) only scaled to about 115 threads.

One of the other results of the paper is that introduction of fine-grained synchronization increases the hardware size by at most 10%, that almost entirely due to the ESSB cache-like structure itself.

I claim this provides additional justification for doing things in a fine-grained manner in parallel computing models, as in Ephemeral, although this paper only addresses synchronization methods.

DOI: http://dx.doi.org/10.1145/1995896.1995948

7.30 Programmable Data dependencies and Placements [7]

This paper is based on the factoring of a (data-independent) program into a data dependency graph, and the individual computations performed at graph nodes. The proposed use of a data dependency algebra (DDA) to generate the dependency graph is described, as well as use of a space-time DDA (STA) to describe the communication topology of a parallel processor architecture. The main idea is that a (data-independent) program can be expessed as the following three modules:

1. The algorithm, independent of data dependency.

2. The data dependency graph expressed as a DDA.

3. The mapping/embedding from the DDA to a the STA of the execution platform.

and that 3 can be generated automatically from 2 and the platform STA, while the programmer factors the program into 1 and 2. The compiler writers are to be responsible for generating the STA graph descriptions of target platforms. The paper goes on to describe DDAs for various parallel algorithms, then STAs for various processing topologies, then various embeddings of DDAs into STAs, and then code generation. The system was not yet implemented, so the performance results are due to manually “compiled” code.

This system is simillar to a proprietary system “GAUSS” that I ‘almost’ implemented in 1989. GAUSS did not require programs to be data-independent, however data-dependent programs required more programmer annotation (manual allocation of processor resources).

It is also simillar to another proprietary system I proposed later in 1989, that would automatically handle C code, limited to the data-independent case.

This paper, together with some of its referrences, shows that automatic placement of computations onto processors in various topologies has long advanced to a point sufficient to support many practical high performance applications.

DOI: http://dx.doi.org/10.1145/2103736.2103741

7.31 Expressive array constructs in an embedded GPU kernel programming language [14]

This paper describes the experimental addition of a secondary kind of array to the Obsidian GPU Kernel Programming Language.

Obsidian is a Haskell package for generating GPU kernels. Using Obsidian, one writes Haskell code that produces a Kernel. Unlike many code generators, the Obsidian/Haskell program resembles the generated kernel, so the programming process is more like writing a kernel in a high level language than like writing a code generator. Obsidian provides GPU-limited versions of the usual programming constructs, including arrays. The existing array construct, "pull" arrays, may be read using complex index expressions, but may only be constructed (written) using coherent indexing, such as an affine function of the threadid. This works quite well sometimes, for example, allowing a simple form of automatic loop fusion. In some cases, such as array concatenation, this is inefficient, as it requires use of conditionals within a kernel. The situation is improved through the use of the novel "push" arrays. Push arrays may be written using complex indexing expressions, but may not be read that way. After construction, a push array may then be effectively converted to a pull array, requiring insertion of synchronization code between these different uses. The paper proceeds to illustrate the use of push arrays via simple parallel sorting kernels (Batcher's bitonic algorithm) and performance measurements.

This is another example of a case where a particular optimization (separation of gather and scatter operations into layers with intervening synchronization), has the following characteristics: One would not expect it to be built into any compiler, but research shows it is quite effective for certain specialized situations. It seems obvious enough to be discoverable by any serious programmer.

DOI: http://dx.doi.org/10.1145/2103736.2103740

7.32 The Sequential Prison [49]

The computer graphics (and OOP) pioneer, Ivan Sutherland, makes the case that the self-propagating cycle of learning and teaching sequential programming has become so entrenched that there will be no escape without a significant change in the way computation is seen. More specifically, he claims that even the use of languages encoded as sequences of characters causes a sequential bias in our thinking about computations so expressed. Many other observations are made in his talk. Asynchronous logic is almost never used in modern systems, even though it potentially offers considerable energy savings. This may be evidence that computer engineers are stuck in a rut (sequential clocked vs. self-timed logic) similar to that (sequential vs. parallel) of computer scientists. The way programming is described contributes to the sequential prison. A program is usually <u>defined</u> as a <u>sequence</u> of steps. Sutherland says that computer scientists need to stop using the word "programming" for what they do, and use another word, such as (perhaps) "configuration", if they are to ever escape the rut.

The current proposal seeks to avoid becoming trapped in the sequential prison, in various ways including the following: Predicate logic programming is used at all levels, so that effort is required to introduce sequencing. Only an abstract syntax is defined, to avoid the necessity of expressing programs as a sequence of bytes. The system target is the Ephemeral language, which itself avoids sequential bias.

DOI: http://dx.doi.org/10.1145/2048066.2048068

VIDEO: http://dl.acm.org/ft_gateway.cfm?id=2048068&ftid=1163270 &dwn=1&CFID=120492121&CFTOKEN=31776533

7.33 Adapt or become extinct! The Case for a Unified Framework for Deployment-Time Optimization [24]

I should perhaps take this advice, as this project was hatched in the early 80s. =)

This paper advocates the adoption of adaptation in many forms as a necessity for future high - performance computing applications, especially for scaling the "walls" of memory, communications, parallel programming, power, etc.

It appears that homogeneity may never arrive in the area of high-performance/high-efficiency computing the way it has for desktop computing. This in mind, the authors point out that a code that is tuned for one system is quite likely to perform poorly on another system, due to potential differences in a variety of system attributes, all of which may need to be accounted for in the code if it is to perform well. These attributes include ISA, number of processors, memory per processor, interconnection network, cache sharing and cache hierarchy, among others.

Various extant adaptation strategies are discussed, such as optimizing compilers, algorithms with various adjustable performance parameters, auto-tuning libraries, choice of different communication packages, scheduling methods, and cache-coherence protocols.

It is also pointed out that sometimes adaptation must be done at run-time, for example choosing matrix representations and algorithms based on the sparseness of data and on the nature of sub-structures. Another example is the use of schedulers that schedule complementary workloads together to optimize resource utilization. Another example is use of profiling data.

It is also pointed out that all these techniques are applied in a rather ad-hoc manner, and that a more holistic approach will be necessary in the future. The authors propose an "adaptation infrastructure" where a single decision maker receives information both at run-time, and before, in the form of programmer annotations, static analysis, profile data etc., and uses all this information to make adaptation decisions of all kinds, both before and during run-time.

I would only point out that having a programming language capable of expressing and controlling compiler optimizations could greatly simplify the task of creating such an infrastructure, especially in the case where correctness is required.

DOI: http://dx.doi.org/10.1145/2000417.2000422

7.34 Implementation of a Hierarchical N-Body Simulator Using The OmpSs Programming Model [41]

This paper describes lessons learned parallelizing Treecode, an N-body gravitation simulator using the Barns-Hut algorithm, for execution on a moderately (4 6-core Xeon with a total of 48 GB) parallel system.

The OmpSs system is an extension of OpenMP, with additional annotations to designate data flow directionality between tasks. OmpSs also adds an extra thread for each processor to manage data-flow-like task synchronization.

The authors consider this algorithm to be a representative example of "irregular scientific applications", in that it solves a problem that could be solved using array computing on a regular grid, but requires less computation, by doing things in a data-dependent manner.

Even on this system with large traditional processors having large memories, the main lesson reported is that finer grained tasks were much better for load balancing, and that processors should include support for task creation and scheduling for larger numbers of smaller tasks to improve processor utilization.

DOI: http://dx.doi.org/10.1145/2089142.2089150

7.35 Communication and Concurrency [33]

This is Milner's famous reference on verification of concurrent systems, which contains the definition of bisimulation I use. It also contains the 'round-robin' scheduler model used in some of my benchmarks in bisimulation research.

7.36 Concurrency and Automata on Infinite Sequences [40]

This is another famous reference on bisimulation, which must be referenced, but which I did not actually read. I copy the abstract here:

> The paper is concerned with ways in which fair concurrency can be modelled using notations for omega-regular languages languages containing infinite sequences, whose recognizers are modified forms of Büchi or Muller-McNaughton automata. There are characterization of these languages in terms of recursion equation sets which involve both minimal and maximal fixpoint operators. The class of ω-regular languages is closed under a fair concurrency operator. A general method for proving/deciding equivalences between such languages is obtained, derived from Milner's notion of simulation.

DOI: http://dx.doi.org/10.1007/BFb0017309

7.37 Graph-Based Algorithms for Boolean Function Manipulation [6]

Due to time constraints, I merely copy the abstract here:

> In this paper we present a new data structure for representing Boolean functions and an associated set of manipulation algorithms. Functions are represented by directed, acyclic graphs in a manner similar to the representations introduced by Lee [1] and Akers [2], but with further restrictions on the ordering of decision variables in the graph. Although a function requires, in the worst case, a graph of size exponential in the number of arguments, many of the functions encountered in typical applications have a more reasonable representation. Our algorithms have time complexity proportional to the sizes of the graphs being operated on, and hence are quite efficient as long as the graphs do not grow too large. We present experimental results from applying these algorithms to problems in logic design verification that demonstrate the practicality of our approach.

DOI: http://dx.doi.org/10.1109/TC.1986.1676819

7.38 MACLISP Reference Manual [34]

This manual contains all that is needed to understand the detailed operation of MACLISP. It was from this manual that I first learned the operation of LISP macros. As LISP operates by eager evaluation, macros provide one of the few ways to simulate lazy evaluation. The evaluator will call the macro (possibly at run-time) using the source code of the un-evaluated arguments, so that the macro may give to the arguments whatever semantics are desired. Proper use of the argument source code by the macro requires care, as evaluating such source may accidentally cause references to the internal variables of the macro, instead of the entity they statically appear to reference. This and other related problems eventually lead to the invention of 'Hygienic' Macro Expansion, discussed next.

7.39 Hygienic Macro Expansion [30]

I viewed the presentation of this paper at the 1986 conference on LISP and Functional Programming. The authors presented a workable solution to the problems of incorrect references that can occur when using macros in non-statically scoped variants of LISP. It was noted that unfortunately, the solution was implemented entirely in the interpreter and other such supporting code, in such a way that the solution was not clearly describable within the language itself, so that a meta-level solution could not be constructed (within user-modified evals presumably) within the language as implemented. Thus, there remained some messiness clouding the meaning of macro expansions.

DOI: http://dx.doi.org/10.1145/319838.319859

7.40 Syntactic Closures [2]

This paper (which I also viewed at the 1988 conference on LISP and Functional Programming), Presented a solution to the messiness problem for the 'extend-syntax' feature of the R^3 Scheme programming language. Again, however, there was unfinished business. It appeared that the language implementation had to manipulate things in a way that programs in the language could not, in order to make things work cleanly.

DOI: http://dx.doi.org/10.1145/62678.62687

7.41 A Confluent Calculus of Macro Expansion and Evaluation [5]

Due to time constraints, I merely copy the abstract here:

> Syntactic abbreviations or macros provide a powerful tool to increase the syntactic expressiveness of programming languages. The expansion of these abbreviations can be modeled with substitutions. This paper presents an operational semantics of macro expansions and evaluation where substitutions are handled explicitly. The semantics is defined in terms of a confluent, simple, and intuitive set of rewriting rules. The resulting semantics is also a basis for developing correct implementations.

DOI: http://dx.doi.org/10.1145/141471.141562

7.42 The C++ Programming Language [47]

Due to time constraints, I merely copy the abstract here:

> Written by Bjarne Stroustrup, the creator of C, this is the world's most trusted and widely read book on C. For this special hardcover edition, two new appendixes on locales and standard library exception safety have been added. The result is complete, authoritative coverage of the C language, its standard library, and key design techniques. Based on the ANSI/ISO C standard, The C Programming Language provides current and comprehensive coverage of all C language features and standard library components. For example: abstract classes as interfaces class hierarchies for object-oriented programming templates as the basis for type-safe generic software exceptions for regular error handling namespaces for modularity in large-scale software run-time type identification for loosely coupled systems the C subset of C for C compatibility and system-level work standard containers and algorithms standard strings, I/O streams, and numerics C compatibility, internationalization, and exception safety Bjarne Stroustrup makes C even more accessible to those new to the language, while adding advanced information and techniques that even expert C programmers will find invaluable.

7.43 The C++ Programming Language [48]

Due to time constraints, I merely copy the abstract here:

> C++11 has arrived: thoroughly master it, with the definitive new guide from C++ creator Bjarne Stroustrup, C++ Programming Language, Fourth Edition! The brand-new edition of the world's most trusted and widely read guide to C++, it has been comprehensively updated for the long-awaited C++11 standard. Extensively rewritten to present the C++11 language, standard library, and key design techniques as an integrated whole, Stroustrup thoroughly addresses changes that make C++11 feel like a whole new language, offering definitive guidance for leveraging its improvements in performance, reliability, and clarity. C++ programmers around the world recognize Bjarne Stoustrup as the go-to expert for the absolutely authoritative and exceptionally useful information they need to write outstanding C++ programs. Now, as C++11 compilers arrive and development organizations migrate to the new standard, they know exactly where to turn once more: Stoustrup's C++ Programming Language, Fourth Edition.

7.44 Preliminary Ada Reference Manual [29]

I learned Ada from this version of the Ada Reference Manual in 1980. It is interesting to remember that generic procedures was in its own separate chapter from non-generic procedures. The use of generics in Ada is quite strongly typed, and appears to be a definite advance over the use of macros when program understandability is of paramount importance. Ada generics are, of course, much less flexible than C++ templates, but, at the time, I saw that the Ada type system was at least more flexible than that of Pascal. The Ada language remained essentially unchanged until the revisions by the Ada9X efforts, which were probably implemented sometime after 1999.

DOI: http://dx.doi.org/10.1145/956650.956651

7.45 The Java Language Specification, Java SE 7 Edition [23]

Due to time constraints, I merely copy the abstract here:

> Written by the inventors of the technology, The Java Language Specification, Java SE 7 Edition, is the definitive technical reference for the Java programming language. The book provides complete, accurate, and detailed coverage of the Java programming language. It fully describes the new features added in Java SE 7, including the try-with-resources statement, multi-catch, precise rethrow, diamond syntax, strings-in-switch, and binary literals. The book also includes many explanatory notes, and carefully distinguishes the formal rules of the language from the practical behavior of compilers.

7.46 C++ Seminar [45]

Due to time constraints, I merely copy the abstract here:

> C++ usage has changed drastically over the past ten years with much more advanced use of templates, meta-programming, functors, and other high-level concepts becoming common. Last year, a new C++ standard (C++11, formerly C++0x) was released with many new and exciting changes to the language that follow these trends.

URL: `http://www.cs.ucr.edu/~cshelton/cppsem.cgi`

7.47 Impact of Economics on Compiler Optimization [44]

Due to time constraints, I merely copy the abstract here:

> Compile-time program optimizations are similar to poetry: more are written than are actually published in commercial compilers. Hard economic reality is that many interesting optimizations have too narrow an audience to justify their cost in a general-purpose compiler, and custom compilers are too expensive to write. An alternative is to allow programmers to define their own compile-time optimizations. This has already happened accidentally for C++, albeit imperfectly, in the form of template metaprogramming. This paper surveys the problems, the accidental success, and what directions future research might take to circumvent current economic limitations of monolithic compilers.

DOI: http://dx.doi.org/10.1145/376656.376751

7.48 Simpler multi-threaded model checking via new foundations for implicit encodings [36]

This document provides an appendix having the text of the proof for canonicty of GDDs, along with the proof that sets represented by GDD-encoded characteristic functions are closed over basic set operations (union, intersection, complement, various cartesian products). The appendices also provide details of BundleUnions, the domain over which GDD-encoded functions are defined. The main document also contains a more grandiose proposal which was abandoned due to schedule constraints.

`URL: http://www.cs.ucr.edu/~mummem/ProposalCD.pdf`

7.49 A Symbolic Algorithm for Optimal Markov Chain Lumping [16]

Due to time constraints, I merely copy the abstract here:

> Many approaches to tackle the state explosion problem of Markov chains are based on the notion of lumpability, which allows computation of measures using the quotient Markov chain, which, in some cases, has much smaller state space than the original one. We present, for the first time, a symbolic algorithm and its implementation for the lumping of Markov chains that are represented using Multi-Terminal Binary Decision Diagrams. The algorithm is optimal, i.e., generates the smallest possible quotient Markov chain. Our experiments on various configurations of two example models show that the algorithm (1) handles significantly larger state spaces than an explicit algorithm, (2) is in the best case, faster than an efficient explicit algorithm while not prohibitively slower in the worst case, and (3) generates quotient Markov chains that are several orders of magnitude smaller than ones generated by a model-dependent symbolic lumping algorithm.

`DOI: http://dx.doi.org/10.1007/978-3-540-71209-1_13`

7.50 Using Edge-Valued Decision Diagrams for Symbolic Generation of Shortest Paths [12]

Due to time constraints, I merely copy the abstract here:

We present a new method for the symbolic construction of shortest paths in reachability graphs. Our algorithm relies on a variant of edge-valued decision diagrams that supports efficient fixed-point iterations for the joint computation of both the reachable states and their distance from the initial states. Once the distance function is it known, a shortest path from an initial state to a state satisfying a given condition can be easily obtained. Using a few representative examples, we show how our algorithm is vastly superior, in terms of both memory and space, to alternative approaches that compute the same information, such as ordinary or algebraic decision diagrams.

DOI: http://dx.doi.org/10.1007/3-540-36126-X_16

7.51 SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual [11]

Due to time constraints, I merely copy the abstract here:

SMART is a software package that integrates various high-level logical (functional) and timing/stochastic (nonfunctional) modeling formalisms (e.g., stochastic Petri nets) in a single modeling study. Each (sub)model is described in a uniform environment and solved using a variety of solution techniques, from symbolic model-checking for temporal logic verification to numerical methods and simulation for performance analysis. Since SMART is intended as a research tool, it is written in a modular way that allows researchers to perform the easy integration of new formalisms and solution algorithms. One of the main strengths of SMART is its emphasis on structural decomposition methods for the efficient storage and analysis of discrete-state models.

URL: http://www.cs.ucr.edu/~ciardo/SMART/

7.52 Symbolic State-Space Generation of Asynchronous Systems Using Extensible Decision Diagrams [51]

Due to time constraints, I merely copy the abstract here:

We propose a new type of canonical decision diagrams, which allows a more efficient symbolic state-space generation for general asynchronous systems by allowing on-the-fly extension of the possible state variable domains. After implementing both breadth-first and saturation-based state-space generation with this new data structure in our tool SmArT, we are able to exhibit substantial efficiency improvements with respect to traditional static decision diagrams. Since our previous works demonstrated that saturation outperforms breadth-first approaches, saturation with this new structure is now arguably the state-of-the-art algorithm for symbolic state-space generation of asynchronous systems.

DOI: http://dx.doi.org/10.1007/978-3-540-95891-8_52

7.53 Symbolic computation of strongly connected components and fair cycles using saturation [55]

Due to time constraints, I merely copy the abstract here:

The computation of strongly connected components (SCCs) in discrete-state models is a critical step in formal verification of LTL and fair CTL properties, but the potentially huge number of reachable states and SCCs constitutes a formidable challenge. We consider the problem of computing the set of states in SCCs or terminal SCCs in an asynchronous system. We employ the idea of saturation, which has shown clear advantages in symbolic state-space exploration (Ciardo et al. in Softw Tools Technol Transf 8(1):425, 2006; Zhao and Ciardo in Proceedings of 7th international symposium on automated technology for verification and analysis, pp 368381, 2009), to improve two previously proposed approaches. We use saturation to speed up state exploration when computing each SCC in the Xie-Beerel algorithm, and we compute the transitive closure of the transition relation using a novel algorithm based on saturation. Furthermore, we show that the techniques we developed are also applicable to the computation of fair cycles. Experimental results indicate that the improved algorithms using saturation achieve a substantial speedup over previous BFS algorithms. In particular, with the new transitive closure computation algorithm, up to 10150 SCCs can be explored within a few seconds.

DOI: http://dx.doi.org/10.1007/s11334-011-0146-3

7.54 Forwarding, Splitting, and Block Ordering to Optimize BDD-based Bisimulation Computation [54]

Due to time constraints, I merely copy the abstract here:

In this paper we present optimizations for a BDD-based algorithm for the computation of several types of bisimulations which play an important role for minimisation of large systems thus enabling their verification. The basic principle of the algorithm is partition refinement. Our proposed optimizations take this refinement-structure as well as the usage of BDDs for the representation of the system into account: (1) *block forwarding* updates in-situ newly refined blocks of the partition, (2) *split-driven refinement* approximates the blocks that may be refined, and (3) *block ordering* heuristically suggests a good order in which the blocks will be refined.

We provide substantial experimental results on examples from different applications and compare them to alternative approaches. The experiments clearly show that the proposed optimization techniques result in a significant performance speed-up compared to the basic algorithm as well as to alternative approaches.

`link: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.65.655`

7.55 Symbolic bisimulation minimisation [4]

Due to time constraints, I merely copy the abstract here:

We describe a set of algorithmic methods, based on symbolic representation of state space, for minimisation of networks of parallel processes according to bisimulation equivalence. We compute this with the Coarsest Partition Refinement algorithm, using the Binary Decision Diagram structures. The method applies to labelled synchronised vectors of finite automata as the description of systems. We report performances on a couple of examples of a tool being implemented.

`DOI: http://dx.doi.org/10.1007/3-540-56496-9_9`

7.56 Approximate steady-state analysis of large Markov models based on the structure of their decision diagram encoding [52]

Due to time constraints, I merely copy the abstract here:

> We propose a new approximate numerical algorithm for the steady-state solution of general structured ergodic Markov models. The approximation uses a state-space encoding based on multiway decision diagrams and a transition rate encoding based on a new class of edge-valued decision diagrams. The new method retains the favorable properties of a previously proposed Kronecker-based approximation, while eliminating the need for a Kronecker-consistent model decomposition. Removing this restriction allows for a greater utilization of event locality, which facilitates the generation of both the state-space and the transition rate matrix, thus extends the applicability of this algorithm to larger and more complex models.

DOI: http://dx.doi.org/10.1016/j.peva.2011.02.005

URL: `http://www.cs.iastate.edu/~ciardo/pubs/2011PEVA-Approx.pdf`

8 Acknowledgements

I acknowledge the Divine Providence of my Lord Jesus, the Christ, especially in showing that the saturation heuristic would work with non-deterministic bisimulation, and lumping in 2009, and for making obvious the canonicity of GDDs. I also thank Him for bringing me to the right advisor at the right time. I thank my advisor for supporting my travel to RP2011, and for suggesting the lumping problem in 2008.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[2] A. Bawden and J. Rees. Syntactic closures. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 86–95, New York, NY, USA, 1988. ACM.

[3] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 390–404, London, UK, UK, 2000. Springer-Verlag.

[4] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In G. von Bochmann and D. Probst, editors, *Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 96–108. Springer Berlin Heidelberg, 1993.

[5] A. Bove and L. Arbilla. A confluent calculus of macro expansion and evaluation. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 278–287, New York, NY, USA, 1992. ACM.

[6] R. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, Aug 1986.

[7] E. Burrows and M. Haveraaen. Programmable data dependencies and placements. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 31–40, New York, NY, USA, 2012. ACM.

[8] M.-Y. Chung, G. Ciardo, and R. I. Siminiceanu. Distributed saturation. Technical report, Hampton, VA, United States, 2007. NASA/CR-2007-214862, NIA Report No. 2007-05.

[9] M.-Y. Chung, G. Ciardo, and R. I. Siminiceanu. Caching, hashing, and garbage collection for distributed state space construction. *Electronic Notes in Theoretical Computer Science*, 198(1):121–136, 2008. Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007).

[10] M.-Y. Chung, G. Ciardo, and A. J. Yu. A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis. In *Proceedings of the 4th International Conference on Automated Technology for Verification and Analysis*, ATVA'06, pages 51–66, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] G. Ciardo and Others. Smart: Stochastic model checking analyzer for reliability and timing, user manual. http://www.cs.ucr.edu/˜ciardo/SMART/.

[12] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In M. Aagaard and J. O'Leary, editors, *Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 256–273. Springer Berlin Heidelberg, 2002.

[13] G. Ciardo, Y. Zhao, and X. Jin. Parallel symbolic state-space exploration is difficult, but what is the alternative? In *PDMC*, pages 1–17, 2009.

[14] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded gpu kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 21–30, New York, NY, USA, 2012. ACM.

[15] R. Cledat, K. Ravichandran, and S. Pande. Leveraging data-structure semantics for efficient algorithmic parallelism. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 28:1–28:10, New York, NY, USA, 2011. ACM.

[16] S. Derisavi. A symbolic algorithm for optimal markov chain lumping. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin Heidelberg, 2007.

[17] J. Ezekiel, G. Lüttgen, and G. Ciardo. Parallelising symbolic state-space generators. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 268–280, Berlin, Heidelberg, 2007. Springer-Verlag.

[18] J. Ezekiel, G. Lüttgen, and R. Siminiceanu. Can saturation be parallelised? In L. Brim, B. Haverkort, M. Leucker, and J. van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *Lecture Notes in Computer Science*, pages 331–346. Springer Berlin Heidelberg, 2007.

[19] J. Ezekiel, G. Lüttgen, and R. Siminiceanu. To parallelize or to optimize? *J. Log. and Comput.*, 21(1):85–120, Feb. 2011.

[20] J. Ezekiel, G. Lüttgen, and R. I. Siminiceanu. An anticipated firing saturation algorithm for shared-memory architectures.

[21] J. Ezekiel, G. Lüttgen, and R. I. Siminiceanu. A parallel saturation algorithm on shared memory architectures.

[22] L. Fix, O. Grumberg, A. Heyman, T. Heyman, and A. Schuster. Verifying very large industrial circuits using 100 processes and beyond. In *Proceedings of the Third International Conference on Automated Technology for Verification and Analysis*, ATVA'05, pages 11–25, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 7 Edition.* Addison-Wesley Professional, 1st edition, 2013.

[24] G. Goumas, S. A. McKee, M. Själander, T. R. Gross, S. Karlsson, C. W. Probst, and L. Zhang. Adapt or become extinct!: the case for a unified framework for deployment-time optimization (position paper). In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 46–51, New York, NY, USA, 2011. ACM.

[25] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Proceedings of the 13 IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*, CHARME'05, pages 129–145, Berlin, Heidelberg, 2005. Springer-Verlag.

[26] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. In J. Hunt, WarrenA. and F. Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 54–66. Springer Berlin Heidelberg, 2003.

[27] G. W. Hamilton and N. D. Jones. Distillation with labelled transition systems. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 15–24, New York, NY, USA, 2012. ACM.

[28] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 20–35, London, UK, UK, 2000. Springer-Verlag.

[29] J. D. Ichbiah. Preliminary ada reference manual. *SIGPLAN Not.*, 14(6a):1–145, June 1979.

[30] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM.

[31] D. Kunkle. Roomy: A system for space limited computations. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 22–25, New York, NY, USA, 2010. ACM.

[32] D. Kunkle, V. Slavici, and G. Cooperman. Parallel disk-based computation for large, monolithic binary decision diagrams. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 63–72, New York, NY, USA, 2010. ACM.

[33] R. Milner. *Communication and Concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[34] D. A. Moon. *MACLISP Reference Manual.* M.I.T. Project Mac, Cambridge, Mass., USA, 1974.

[35] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 117–120, New York, NY, USA, 2012. ACM.

[36] M. Mumme. Simpler multi-threaded model checking via new foundations for implicit encodings, 2014. http://www.cs.ucr.edu/˜mummem/ProposalCD.pdf.

[37] M. Mumme and G. Ciardo. A fully symbolic bisimulation algorithm. In *Proceedings of the 5th International Conference on Reachability Problems*, RP'11, pages 218–230, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] M. MUMME and G. CIARDO. An efficient fully symbolic bisimulation algorithm for non-deterministic systems. *International Journal of Foundations of Computer Science*, 24(02):263–282, 2013.

[39] M. Odersky. *The Scala Language Specification.* École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, 2011.

[40] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, UK, 1981. Springer-Verlag.

[41] M. Pericàs, X. Martorell, and Y. Etsion. Implementation of a hierarchical n-body simulator using the ompss programming model. In *Proceedings of the first workshop on Irregular applications: architectures and algorithm*, IAAA '11, pages 23–30, New York, NY, USA, 2011. ACM.

[42] B. Pradelle, P. Clauss, and V. Loechner. Adaptive runtime selection of parallel schedules in the polytope model. In *Proceedings of the 19th High Performance Computing Symposia*, HPC '11, pages 81–88, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[43] J. Ributzka, Y. Hayashi, J. B. Manzano, and G. R. Gao. The elephant and the mice: the role of non-strict fine-grain synchronization for modern many-core architectures. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 338–347, New York, NY, USA, 2011. ACM.

[44] A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, JGI '01, pages 1–10, New York, NY, USA, 2001. ACM.

[45] C. Shelton. C++ seminar, 2012. http://www.cs.ucr.edu/˜cshelton/cppsem.cgi.

[46] T. Stornetta and F. Brewer. Implementation of an efficient parallel bdd package. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC '96, pages 641–644, New York, NY, USA, 1996. ACM.

[47] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[48] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[49] I. Sutherland. The sequential prison. *SIGPLAN Not.*, 46(10):1–2, Oct. 2011.

[50] V. Ureche, T. Rompf, A. Sujeeth, H. Chafi, and M. Odersky. Stagedsac: a case study in performance-oriented dsl development. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 73–82, New York, NY, USA, 2012. ACM.

[51] M. Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In M. Nielsen, A. Kučera, P. Miltersen, C. Palamidessi, P. Tůma, and F. Valencia, editors, *SOFSEM 2009: Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 582–594. Springer Berlin Heidelberg, 2009.

[52] M. Wan, G. Ciardo, and A. S. Miner. Approximate steady-state analysis of large markov models based on the structure of their decision diagram encoding. *Performance Evaluation*, 68(5):463 – 486, 2011.

[53] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Active pebbles: parallel programming for data-driven applications. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 235–244, New York, NY, USA, 2011. ACM.

[54] R. Wimmer, M. Herbstritt, and B. Becker. Forwarding, splitting, and block ordering to optimize bdd-based bisimulation computation. In C. Haubelt and J. Teich, editors, *Proceedings of the 10^{th} GI/ITG/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen" (MBMV)*, pages 203–212, Erlangen, Germany, mar 2007. Shaker Verlag.

[55] Y. Zhao and G. Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *Innovations in Systems and Software Engineering*, 7(2):141–150, 2011.

[56] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.