Verification of Vortex Workflows

Xiang Fu[1] Tevfik Bultan[1] Richard Hull[2] Jianwen Su[1]

[1] Department of Computer Science, University of California, Santa Barbara, CA 93106, USA. {fuxiang, bultan, su}@cs.ucsb.edu.
[2] Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974. hull@research.bell-labs.com.

Abstract. Vortex is a workflow language to support decision making activities. It centers around gathering and computing attributes of input objects. The semantics of Vortex is declarative, and the dependency graphs of Vortex programs are acyclic. This paper discusses the application of symbolic model checking techniques to verification of Vortex programs. As a case study we used a Vortex program MIHU for online customer support. The control structure and the declarative semantics of Vortex programs enabled us to develop various optimization techniques for the purpose of verification. These techniques include constructing a disjunctive transition BDD, variable pruning, projection of initial constraints, and predicate abstraction.

1 Introduction

A workflow management system provides a mechanism for organizing the execution of multiple tasks, typically in support of a business or scientific process. A variety of workflow models and implementation strategies have been proposed [8, 19]. Workflows concentrate on the control and coordination of tasks performed by software systems or humans. Workflows are typically represented using some form of directed graphs [8, 6], often based on variations of Petri nets. A workflow specification describes both data and control flows between tasks as well as the application programs that implement the tasks. In contrast, recent work in the area of scientific workflows emphasize the need to promote a dataflow view of the workflow at the specification level. [1] presented an "object view" where the focal point is the data used and generated during workflow execution. There, workflows are considered as graphs of objects with the processes that created them being expressed through the links between them. A mixed view is proposed in [14], wherein both data and control flow are expressible and the user can navigate from an activity to its input data structure.

Many workflow applications require highly differentiated treatments for different kinds of inputs. A substantial class of examples arises in customer care (e.g., e-commerce, insurance claims processing) where enterprises attempt to provide goods and services to a mass market. Such individualized treatments can cater to the individual preferences of customers, and can support targeted marketing initiatives and promotions by the host enterprises. This is especially important in connection with establishing and maintaining a loyal customer base, a cornerstone to success in business in general and e-commerce in particular.

To support this need, Vortex [10] is developed recently and focuses on determining attribute values, and the means by which these values are obtained are modeled essentially as side-effects of this. However, attributes in Vortex can either model the execution status of some task or the output data of a task. This approach offers a lot of flexibility, which enables to alternatively focus on tasks or data as needed by the decision process incarnated by the workflow. Vortex enables a declarative specification of workflows, which matches the need for high-level specification languages for designing and prototyping workflows usable by non computer trained users (e.g., a natural scientist) emphasized by existing work on scientific workflows.

Increasingly more people are getting information and services online, many of which are supported by workflow systems. Failure of these systems will have potentially a huge impact (e.g., the server attacks on CNN and other Internet sites some time ago). Moreover, workflow specifications (programs) are becoming more and more complex. For example, a Vortex workflow program [10] in practical use may consist of hundreds of variables and thousands lines of code. It is unlikely that one can develop large workflow systems free of errors with no tools. An interesting issue here is to develop appropriate tools to aid the design of workflow specifications. Good design tools can not only improve the quality of workflow specifications and but also improve the design and maintenance process. Verification techniques can allow the designer to "debug" the specifications of workflow processing logic. Among important properties of workflow specification are logical properties such as each insurance claim is eventually approved or disapproved, existence of nondeterministic behaviors, and properties related to tasks with side effects (such as issuing a check).

Research on model-checking produced tools such as SMV [13] has been successfully applied to verification of control-intensive systems [12]. Symbolic model checking has been used in verification of systems with up to 10^{100} states. However, large number of variables, complex program logic, or arithmetic operations can easily exceed the capabilities of verification tools such as SMV. The control structure and the declarative semantics of Vortex programs provide opportunities for various optimizations which can result in scalable verification techniques.

In [15], modeling checking was applied to verification of Mentor workflow specifications. More specifically, the focus is on properties over graph structures (rather than execution results). A similar approach was taken using Petri-net based structures in [18]. A technique for translating business processes in the process interchange format (PIF) to CCS was developed in [17] which can then be verified by appropriate tools. Clearly, a direct verification that considers not only the structures but also the executions is more accurate and desirable. This is the focus of the present paper.

In this paper we present techniques such as constructing a disjunctive transition BDD, variable pruning, projection of initial constraints, and predicate abstraction for verification of Vortex programs. As a case study we use a Vortex program "May-I-Help-You" (MIHU), a system to improve the effectiveness of web-based storefronts. MIHU has over 40 integer attributes and consists of more than 800 lines of Vortex code. A straightforward mapping of the MIHU program to SMV results in a BDD of size too large to be computed. By introducing execution order on Vortex programs we were able to construct a much smaller

disjunctive transition BDD in 10 seconds. However, even using the disjunctive transition BDD we were not able to check all the properties of the MIHU program. Based on the dependencies among attributes we were able to develop a variable pruning technique which only preserves the variables that are *active* during the computation. This technique is motivated by the acyclic nature of dependencies in Vortex programs and can be applied to other such systems. Variable pruning requires the projection of the initial image to different stages of the computation. This also reduces the size of the initial image, since complicated predicates on various attributes, such as sorted arrays, are decomposed. Using these techniques we were able to check all the properties of the MIHU program using SMV.

The remainder of the paper is organized as follows. § 2 reviews necessary concepts of Vortex. § 3 describes a Vortex application system "May I Help yoU" (MIHU), which we use as a case study. § 4 gives the mapping from Vortex to SMV and compares two different approaches to construct an efficient transition BDD. § 5 presents two optimization methods: variable pruning and decomposition of initial constraints. § 6 is an experimental exploration on predicates abstraction, which sheds a light on solving the exponential problem size over integer width.

2 Vortex Decision Flows

Vortex [10] is a programming paradigm for online decision making, an important component in workflow systems. A Vortex program focuses on information gathering, decision making, and launching and monitoring of external tasks. The Vortex language is declarative and it allows programmers to specify the conditions under which decisions and tasks should be performed, but the flow of control is not specified explicitly. As a result, Vortex programs are more succinct and easier to analyze formally than equivalent procedural programs, and are easier for humans to understand and modify. Because the core semantics of Vortex is declarative, analysis is simpler than with procedural workflow languages.

A Vortex decision flow consists of a family of attributes that may be evaluated during execution. One attribute will be "target" and embody the output of a decision flow, e.g., what priority of service to give this customer, or what promotional image to display on the next web page. Other attributes correspond to intermediate results of the decision flow. For example, a "promo hit list" attribute might hold a listing of potential promo messages to display, along with scores combining the likelihood that a customer will buy the promo and the potential profit that might be derived. Some intermediate attributes might gather data from external sources, such as databases. Since attribute evaluation can have a real cost, enabling conditions are used to decide which attributes should be evaluated. The set of data flow and control flow dependencies in a decision flow must be acyclic. This and the enabling conditions restrict Vortex decision flows to be "monotonic": once an attribute obtains a value the value will not be changed before the end of execution. This "attribute-centric" perspective of decision flows permits a systematic approach for specifying what factors to be incorporated as a decision is being made.

Individual Vortex programs are centered around the decision making and processing needed to react to a single event, e.g., a new claim input to an in-

surance claims workflow, or a customer contact through a web-based storefront. Programmers specify what tasks might potentially be performed for an incoming event, and specify logical conditions under which these tasks should be performed. A typical Vortex application will involve several Vortex programs, each for dealing with a different class of events.

Programs in Vortex focus on how values should be assigned to the attributes of an input object. In particular, each Vortex execution begins with values only for the *source* attributes. As execution continues values for additional attributes are obtained, perhaps by computation or synthesis based on previously obtained attribute values, by information retrieval, or by interaction with humans or other software systems. Not all attributes need take values. External actions (e.g., issue checks) may be launched as a side-effect of attribute evaluation.

A Vortex program includes *enabling conditions* to determine whether an attribute will be evaluated for a particular execution. These are similar to the enabling rules in Meteor [11] with a crucial difference: the enabling rules of Meteor explicitly mention events, and can be fired only if the mentioned event(s) occur and the remainder of the condition is true at that time. Thus, an analysis of tasks and enabling rules in Meteor requires an understanding of the relative timing of tasks executions. In contrast, the assignment of attribute values in Vortex is monotonic (i.e., once assigned an attribute value cannot change), and the enabling conditions refer only to attribute values and states (i.e., whether an attribute has been enabled or disabled). This makes it possible for Vortex to have a simple declarative semantics that ignores issues around order of execution except for those implied by data flow constraints between tasks.

Thus the computation of an attribute value in a Vortex program may depend on values of other attributes (*data dependencies*) and the execution of the computation depends on attributes occurring in the enabling condition (*control dependencies*).

The attribute-centric paradigm and assumption of monotonicity make possible a natural mechanism for querying the status and history of workflow processing. This is based on the notion of *snapshot* of a Vortex execution. Suppose a Vortex program is being executed. At a given point in time the snapshot associated with this execution is a mapping from attributes to the current state of the attribute (not-yet-considered, enabled or disabled), and if the attribute is enabled either a value for the attribute or the "value" `uninitialized`. Once an execution completes then its snapshot can be archived.

Although attribute computation in Vortex can be specified in several different ways including by external systems (black boxes), an interesting class of computation is specified by *decision modules*. These provide an eclectic mix of mechanisms for aggregating and synthesizing previously obtained attribute values. As a very simple illustration, suppose that multiple vendors are being considered in connection with a given purchase. Different factors might need to be weighed, e.g., the price quoted by different vendors, the availability date, previous history with the vendor, and etc. A Vortex decision module (or family of decision modules) might be used to associate a weight to each vendor and then pick the vendor with highest weight. As a simplified illustration, a rule such as "If vendor V gave lowest price quote, then contribute $[V, 10]$" can be used to contribute a weight of 10 in favor of V; and the total weight for V would be ob-

tained by adding this with the other weights contributed for V. Vortex supports a broad family of simple and intricate semantics for combining the contributions of the rules in a decision module. Decision modules provide a simple mechanism for using a broad variety of heuristics when combining information. This is useful in contexts that involve business considerations, e.g., customer care, automated resolution of exceptions, and reconciling dirty data.

3 A Vortex System "May I Help yoU" (MIHU)

Need for online decision making arises frequently in electronic commerce applications. For example, many web storefronts have human agents for online customer support, in case when customers feel lost while shopping at the website. However in such an application the web storefront has to automatically decide who needs the service and when it is appropriate to prompt a customer to launch the service. A decision process behind the web server can track each shopping session, collect data, analyze the status of a customer, and make decisions to provide online support. MIHU is such a Vortex workflow that runs behind web server to improve customer satisfaction.

The purpose of MIHU is to make the decision whether to provide a customer service called AWD (Automated Web Dialog). The target attribute is a boolean attribute `offer_AWD`, as shown in Fig. 1. Each time a web page is accessed, MIHU will be called by the web server before the page is delivered to the customer. In each run of MIHU, it will monitor several key attributes of a customer (e.g. the business value, the frustration score, and the opportunity score of the current session, and the current agent load of the web store). The goal is to respond as soon as possible whenever the customer's frustration level becomes high or the customer has potential to buy more. As shown in Fig. 1, MIHU will also compute some intermediate attributes such as `AWD_score` and `AWD_override_score` during its execution. The target attribute `offer_AWD` is computed using `AWD_score`, `AWD_override_score` and `agent_load`. Note that there are many source attributes in the program, such as `card_color`, `log`, and `shopping_count` etc. The source attributes are passed by the web server or fetched from databases. The control and data dependencies between all attributes are also shown in Fig. 1, where the graph is acyclic as explained in Section 2.

One characteristic of Vortex programs is that they are heavily "control intensive." A great proportion of a Vortex program consists of case-statements which are specified as conditions on attributes. Let us take a look at a module in MIHU which computes the attribute `frustration_aggregate`.

```
MODULE compute_frustration_aggregate{
enabling condition: true;
computation: frustration_aggregate=
  eval_rules(
    //if no rule is true, the default value is 0
    policy: max_of_true_rules(0),
    rules:{
      if (sorted_vector[1] >= 100) then contribute
         2*(sorted_vector[0]+sorted_vector[1]+sorted_vector[2]);
      if (sorted_vector[0] >= 100) then contribute
```

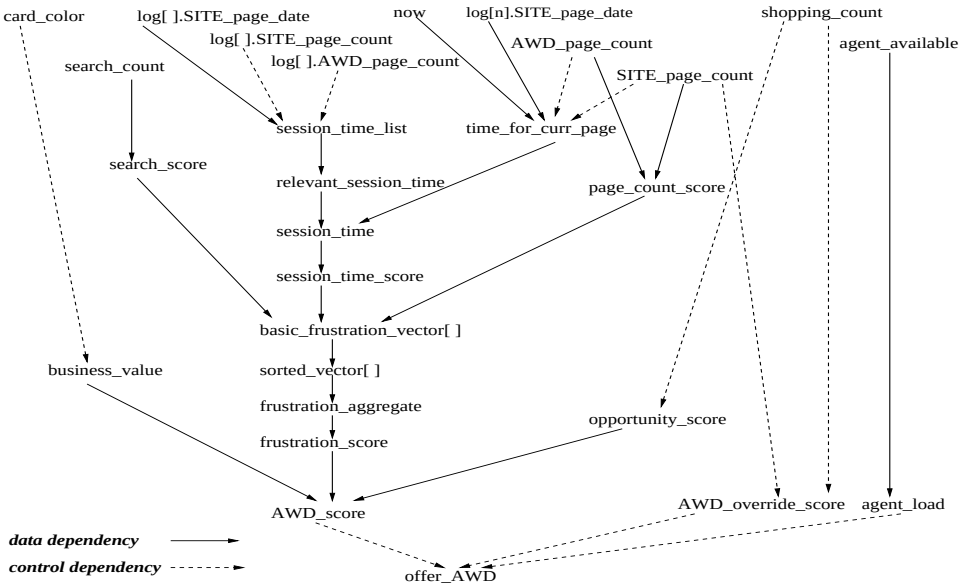

Fig. 1. Dependency Graph for MIHU

```
        (sorted_vector[0]+sorted_vector[1]+sorted_vector[2]);
    if (sorted_vector[0] < 100) then contribute
        (sorted_vector[0]+sorted_vector[1]/2+sorted_vector[2]/4);
  }
 )
}
```

There are three case statements in the module, referred as *rules* in Vortex. The three cases are all evaluated, and the maximum result value is assigned to the attribute `frustration_aggregate`. If none of the cases are satisfied, the attribute is assigned a default value. Note that each Vortex module has an enabling condition (e.g. the above module has an enabling condition which is `true`).

A collection of properties were proposed on MIHU, so that commercial policy and integrity can be ensured. Some of them are listed in the following.

1. AWD should not be provided when there are no human agents for support.
2. MIHU should not provide AWD when one AWD has already been launched.
3. Do not show AWD to any user who has been idle for two hours or more.
4. No AWD should be displayed within the first three pages in any session.

We define P as: there are no human agents for support, or at least one AWD has already been launched, or the user has been idle for more than two hours, or no more than three pages has been displayed. If P is `false`, then the following properties should hold.

5. MIHU should guarantee that the frustration score of any customer is below 90, or an AWD should be launched.
6. When a customer has more than 5 items in his shopping cart, an AWD frame should be launched (to encourage shopping).

7. When a customer has searched the same item for over 5 times, an AWD frame should be launched to help him.

These properties can be expressed as invariants using the Vortex attributes. For example, property (1) can be expressed as `AG( IsEndOfExecution -> ( agent_available==0 -> !offer_AWD) )`, using the CTL operator AG.

4 From Vortex to SMV

It is not unusual for a Vortex workflow system to have many integer variables (e.g., more than 50). Assuming 50 integer variables with 16-bit integer width, the state space would exceed 10^{200}. Explicit exploration on such a huge state space would be computationally very expensive. On the other hand, BDD-based symbolic model checking can cope with large state spaces, given that they can be represented compactly using BDDs. It is true that in the presence of nonlinear constraints BDD representation is not efficient. However a large class of workflow and decision flow used on applications is linear. MIHU is such an example. Therefore we chose the BDD-based symbolic model checker SMV to investigate the feasibility of automated verification of Vortex programs.

4.1 Straightforward mapping

Specification constructs provided in SMV input language enables a straightforward translation of the Vortex programs. Since SMV does not have global variables, a module named `Attributes` is set up to hold attributes, and passed as an argument to each computation module. Any attribute in Vortex program is mapped into one or a set of boolean variables, depending on whether it is a boolean or an integer attribute. Also for each attribute $attr_i$, a boolean variable $attr_i\_enabled$ is set up to represent the status of $attr_i$. Notice that since integer attributes are translated into a set of boolean variables in SMV, the arithmetic computation over them should be translated to a series of logical relations over those boolean variables. Since SMV's mapping of integer variables is inefficient, we used a macro developed by Chan for this purpose [4].

As shown in Fig. 2, for each module in Vortex, the mapped SMV module contains two assignments, one for the attribute, and the other is for the status variable. The two assignments are both case-statements,and each rule in Vortex modules corresponds to a case of the case-statement. The construction of the condition for each case varies according to different rule policies provided in Vortex language such as `first_rule_win` and `max_rule_win`. The methodology to build the condition is the same. For example, if the policy is `first_rule_win`, the condition for each case in the SMV case statement is constructed by conjoining the enabling condition of the module with the rule condition and another condition called `anc_enabled`. The boolean condition `anc_enabled` is defined as the conjunction of all status variables of attributes that are direct predecessors of the attribute that is being computed in the dependency graph. Another example on policy `max_rule_win` is shown in Fig. 2. Notice that the first case states that when the attribute has already been assigned its value, it should keep that value. This helps reduce the size of transition BDD.

```
// anc_enabled: all direct predecessors of attr have been computed
// IsMax(res): among all available results, res is the largest.
// It is defined as follows:
//  (cond1 ->(res>=r1)) & (cond2 ->(res>=r2)) &...& (condn ->(res>=rn))
```

Vortex Module

```
MODULE compute_attr
enabling condition: e_cond;

attr=eval_rules{
  policy: max_rule_wins,

  rules:{
   //rule1
   if cond1 then contribute r1;

   //rule2
   if cond2 then contribute r2;

   ...
   //rule n
   if condn then contribute rn;

  }
}
```

SMV Module

```
MODULE compute_attr

 ASSIGN
 next(attr):= case
  //if already assigned, keep the value
  attr_enabled:attr;

  //rule 1
  e_cond & anc_enabled & cond1 &
  IsMax(r1):r1;

  //rule 2
  e_cond & anc_enabled & cond2 &
  IsMax(r2):r2;

  //rule n
  e_cond & anc_enabled & condn &
  IsMax(rn):rn;
 esac;

 //now for status variable
 next(attr_enabled):= case

   e_cond & anc_enabled: 1;
   1:                    0;//default

 esac;
```

Fig. 2. SMV Translation of a Vortex Module

4.2 Sequential execution

Unfortunately experimental results have shown that the straightforward mapping is inefficient. It took more than 24 hours of CPU time to build the transition BDD of 10-bit integer width for MIHU even without verifying any properties. The reason is that, given a BDD variable order in which all integer bits are interleaved, building a monolithic transition BDD for multiple computations can be exponentially expensive. For example, if there are two computation $c := a + b$ and $f := d + e$ that can be executed in parallel, then there are two options to build the transition BDD. One is to build the monolithic transition BDD that incorporates both computations. The other is to introduce a sequence number, and split the computation in two steps. If the integer width is 16, given an interleaved integer variable order, a single addition BDD is 131 nodes, the transition BDD by first approach is 506 nodes, and the BDD by second method is 266 nodes. Obviously, the second approach, to split the computation in sequential steps, is better.

It might be argued that if we rearrange `a`, `b`, `c` and `d`, `e`, `f` in two groups , and interleave them separately, a much smaller transition BDD can be generated for the first approach. It is true for this little example. However it is not applicable to Vortex programs. Since in a Vortex program, all attributes contribute to the target attribute, any two attributes are logically related. This prevents a grouping of variables based on the dependency. Therefore, a natural BDD variable order for Vortex programs is to interleave all integer variables together.

For such a BDD variable order, the straightforward translation fails because it incorporates too many computations in one step.

In our straightforward mapping, the set of status variables for attributes dictates the execution order. Each snapshot (the values of status variables) will decide which module can be executed. However, multiple modules can be executed in one step, which finally leads to an unreasonably large transition BDD. Our improved approach is quite simple. Just replace those status variables with a sequence variable explicitly, and make sure that at each step, only a small number of computations can be executed. Generally, we assign one sequence number for each Vortex module. If a module contains too much computation, then several sequence numbers can be assigned to a single module.

Fig. 3 shows the structure of the transition relation BDD using sequence numbers. The SMV translation in Fig. 3 is similar to the one in Fig. 2 except that the conditions on status variables are replaced by conditions on sequence numbers.

```
SMV Module
// suppose the sequence number assigned
// to this module is 20
MODULE compute_attr
ASSIGN
  next(attr):=case
//keep the value, if it is already assigned
seq>20 : attr
// case 1
seq==20 & enabling_cond & cond1: res1;
// case 2
seq==20 & enabling_cond & cond2: res2;
       :
       :
// case n
seq==20 & enabling_cond & condn: resn;
esac;
```

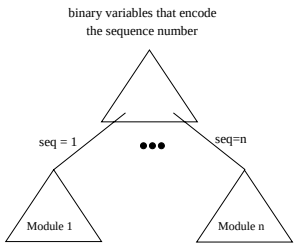

Fig. 3. Disjunctive Structured Transition BDD

As shown in Fig. 3, the transition BDD has a disjunctive structure. So the overall transition BDD size is at worst the sum of all these submodules. Notice that these submodules can share components, which would further reduce the overall transition BDD size. Given a fixed integer width, if all the computations are linear, the total transition BDD size is linear on the program size. It can also be proved that the size is linear on the integer width.

Because of the declarative semantics of Vortex programs, if an execution sequence satisfies the partial order defined in the dependency graph, the final snapshot of all attributes of SMV translation will be the same as the original SMV program. Note that all properties are expressed over the final values of attributes. It is clear that restricting to a particular execution sequence does not affect the verification.

4.3 BDD variable order

The BDD variable order we use in mapping Vortex programs to SMV modules is shown in Fig. 4. The first part is all the bits of the sequence variable `seq` which holds the current sequence number. This part leads to the disjunctive transition BDD structure. As shown in Fig. 4, in the second part we list all the boolean variables that correspond to boolean attributes in Vortex. Finally, the third part

corresponds to the boolean variables that encode the bits of integer attributes. Note that if the ranges of integers are not the same, then we can interleave starting from the least significant bit.

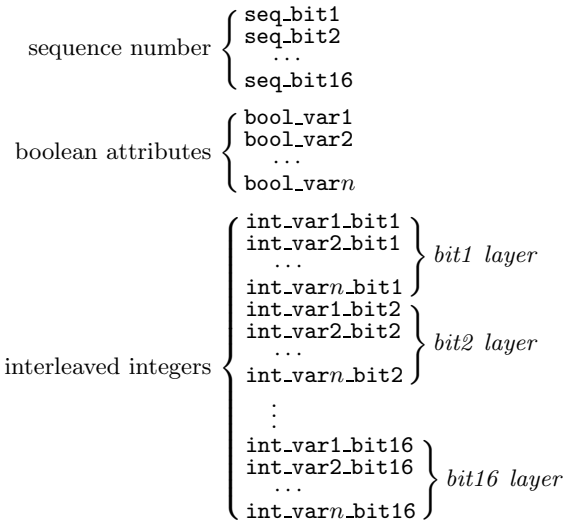

Fig. 4. BDD Variable Order

In Fig. 4, the ordering of variables within each bit layer can affect the size of transition BDD. In any submodules, there are two kinds of computations: the "main computation" of that step, and the assignments to preserve old values for some attributes. If there are too many such attributes, preserving assignment can also result in quite a large BDD, especially when these attributes were mixed with the attributes for main computation. It is better to move these unrelated attributes outside the group of attributes used in main computation, as illustrated in Fig. 5. (In Fig. 5 the edges toward BDD node 0 are all omitted. Attributes u1, u2 are unrelated attributes for the main computation; and r1, r2, r3 are attributes used in computation. The attributes with quotation (e.g. r3′) are next values.) Therefore, heuristically within each bit layer any attributes directly related in dependency graph should be placed close to each other to minimize the number of unrelated attributes in the main computation for each module.

Using the ideas presented above the transition BDD for MIHU Vortex program can be built in 10 seconds for 10 bit integer representation. This is a significant improvement from the straightforward mapping.

5 Optimization

Although the techniques presented in Section 4 enabled us to construct a BDD representation for the Vortex program in a reasonable time, the SMV model checker was not able to verify most of the properties presented in Section 3 using this transition BDD. In this section we present two techniques which enabled us to verify all the presented properties.

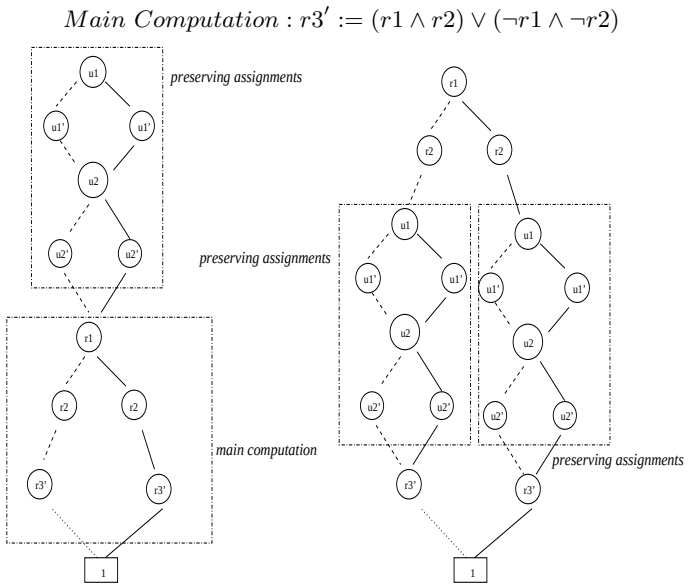

Fig. 5. Effects of Different Variable Order in Bit Layer

5.1 Variable pruning

According to the acyclic property of Vortex, there should be no loop in the execution of any Vortex program. This means that every attribute has a "lifespan",i.e., before its first reference and after its last reference, an attribute is of no use for the computation. Note that for intermediate attributes, "first reference" is their definition; for source attributes, "first reference" is the first time they are used in computation. Naturally an attribute can be pruned outside of its life span – we can treat such variables as "don't cares" by assigning them any possible value nondeterministically. This would greatly reduce the "preserving assignments" cost in sub-BDDs for each module. For example, there are 40 integers in MIHU, so in many modules, there would be over 30 preserving assignments. But after we pruned unnecessary variables by nondeterministic assignment, at any time, there are only less than 9 preserving assignments. For image computation, the improvement is even much greater. There is no need to represent the state of inactive variables; at any time, only those active variables will be represented in image. So actually, the problem size is reduced from a huge state space over all attributes to the state space over the variables that are active. Usually the number of active variables is much smaller than the total number of variables,e.g., in MIHU the maximal number of active variables during any part of computation is 9. Therefore, after pruning inactive variables, we can achieve a much better performance than the original naive method.

5.2 Decomposition of Initial Image

Similar pruning can be applied to source attributes. They can be assigned any value nondeterministically outside their lifespan. What is more, the initial con-

straints can be projected to different stage of execution, which reduces the size of initial image drastically.

The process to project initial image is as follows. Suppose that the initial constraint is expressed in the conjunctive form of $\wedge f(x_i, x_j)$, where $f(x_i, x_j)$ is the logical relation between two attributes. Then for each $f(x_i, x_j)$, suppose the lifespan for the two attributes are $[a_i, b_i]$, $[a_j, b_j]$, and $a_i \geq a_j$. We can map the $f(x_i, x_j)$ into the module of step $a_i - 1$. To do this, transition $f(next(x_i), x_j)$ or $f(next(x_i), next(x_j))$ (if $a_i == a_j$) is added to module $a_i - 1$.

For example, suppose we have an initial constraint over array `log[ ]` that `log` must keep increasing, i.e. `log[0]` $<$ `log[1]` $<$ `log[2]` $<$ `log[3]`. Suppose that `log[2]` is first referenced in module 5, and `log[1]` is first referenced in module 7. Then the constraint `log[1]` $<$ `log[2]` can be mapped into module 6. The program is presented as follows.

```
MODULE No.6
//orginal assignments

TRANSITION
        next(log[1]) < log[2]
```

Notice that the "next" indicates that the next value of `log[1]` should satisfy this constraint in module 7. However the value of `log[2]` will not change, since it is already assigned before module 5.

Since the constraints over arrays can commonly arise in practical applications, (e.g., the date stored in history log should always keep increasing), this kind of constraints brings great trouble for initial image, and lead to quite large initial image size. We have three such kind of constraints over arrays in MIHU, which leads to an initial image of 65,000 BDD nodes, when integer width is 10. The time used to compute the first forward image computation takes more than one hour. After the decomposition optimization applied, the initial image shrinked from 65,000 BDD nodes to 4,000 nodes, and the first forward image computation reduces drastically to only half a second.

5.3 Execution sequence

As we discussed above, the size of active variable set critically affects the cost of verification, i.e., the verification time would increase exponentially over the size of active variable set. The smaller the active set, the smaller the verification cost. Note that different execution sequences lead to different active variable set, e.g., the size of the largest active set in MIHU ranges from 9 to 17 according to different execution sequences. Therefore, it is important to find an optimal execution sequence which minimizes the largest active variable set.

We formalize the notion below to be used in our discussion. Let G be a given dependency graph.

Execution sequence: suppose the total number of attributes is n. For any attribute `attr`, it is mapped to a unique integer in range $[1..n]$. The mapping should satisfy the partial relation defined in dependency graph G.

Lifespan: For each attribute `attr`, its lifespan is a range $[i, j]$, where i is `attr`'s sequence number assigned in execution sequence, j is the maximum sequence number among all its successors.

AVS_i (Active Variable Set at step i): the set of variables active at step i, i.e., $AVS_i = \{\texttt{attr} \mid i \in \texttt{attr}\text{'s lifespan}\}$.

Minimum Active Variable Set (MAVS) problem: Find the execution sequence which minimizes the maximum value for $|AVS_i|$, $1 \leq i \leq n$.

Though the worst case complexity is exponential, a branch and bound algorithm can be used to solve the MAVS problem. Since the MAVS problem is actually to find out an optimal topological order for a directed acyclic graph, during the depth-first search to enumerate all possible topological sequences, we can set a bound on the optimal solution currently available and prune branches that exceed the bound. In addition,greedy search can be started first to find a "not bad" value to initialize the bound. Unfortunately, polynomial time algorithms for MAVS seems unlikely as we can show that the problem is actually NP-hard by a reduction from a known NP-complete problem called *register sufficiency problem* [7].

5.4 Experimental results

We were able to check all properties for the MIHU example given in Section 3 using the techniques discussed above. The hardware we used is a Pentium 450MHZ PC with 128MB memory. Properties 1 and 3 were not satisfied. The errors were due to missing bounds on some of the attributes. For example, `AWD_score` was not bounded in the program which resulted the system assigning an agent to a customer though none were available. The experimental results are as follows.

Integer attributes: 40

Source code in Vortex program MIHU: 800 lines

Maximal active variable set: 9 attributes

Integer Width	Time (Seconds)	Transition BDD Size	Memory Used (Mb)
10	250	269,241	32
11	360	312,548	37
12	560	356,936	39
13	900	401,907	40
14	1500	446,821	42
15	4500	539,843	45

We verified the properties using forward state exploration. First we generated all the reachable states, and verified all the invariants at once. In the MIHU example the most costly computation is the sorting operations to compute `sorted_vector`. It could be possible to improve sorting in such cases using predicate abstraction.

6 Predicate Abstraction

It is not unusual for the complexity of verification to increase exponentially with the integer width. None of the optimization approaches mentioned above can solve this problem. However, the combination of predicate abstraction [9, 5] and decision procedure can help in this problem. Any predicate can be mapped into a boolean variable, where this boolean variable will get assigned at the last module

which defines its related attributes. If with the aid of a decision procedure, we can conclude that this boolean variable can be assigned deterministically 0 or 1, the abstraction on the predicate is complete. Or else we can either leave it nondeterministically assigned, or continue to generate new predicates that leads to a deterministic assignment and spread out more predicates. It is proved that for the nondeterministic abstraction [16] , $\forall$CTL* properties hold in all execution paths, will be preserved, which means that CTL* properties verified to be correct in abstracted version will be correct in the original version; for the deterministic abstraction, in addition to CTL* preservation, properties proved to be incorrect in the abstracted system will not hold in concrete system either.

We tried nondeterministic abstraction in MIHU, which generated 120 new boolean predicates. With this abstraction, all correct properties are verified in 10 minutes. However for the properties that were not verified, the abstraction has to be refined to eliminate false negative results in error trace. This process should be repeated until a fully deterministic abstraction is generated[16]. To generate the fully deterministic abstraction, we estimate that there will be over 500 new predicates for the MIHU.

One bottleneck in the deterministic abstraction method is that: it is required to enumerate the boolean combination over all boolean variables so that

$$InitialImage \Rightarrow R(\bigwedge f(b1, ..., bn))$$

where $f(b1, ..., bn)$ is a boolean expression on boolean variables b1,...,bn (which represent abstracted predicates), and R() is the inverse of abstraction. The goal is to generate a conjunction which is minimal and satisfies the above constraint. The best known complexity of enumeration algorithm to generate such a boolean expression is 3^k [16], where k is the number of all predicates. If k is quite large, the overhead of abstraction would be expensive.

7 Conclusions

In this paper we presented applications of model checking to verification of Vortex workflow programs. We develop techniques for mapping Vortex programs to BDDs efficiently based on the control structure and the declarative semantics of the language. Using these techniques, we were able to verify the Vortex program used for online customer support. We showed that this program did not satisfy all of the stated properties due to missing assumptions on some of the attributes.

We plan to investigate using other symbolic representations such as arithmetic constraints [3] or composite representations [2] for verification of Vortex programs. We will also do more experiments on abstraction techniques, and develop a customized model checker for Vortex language.

Acknowledgements: The work is partially supported by a faculty research grant from UCSB. The work by Bultan is also supported in part by NSF grant CCR-9970976 and NSF CAREER award CCR-9984822. The work by Su is supported in part by NSF grants IRI-9700370 and IIS-9817432.

References

1. A. Ailamaki, Y. Ioannidis, and M. Livny. Scientific workflow management by database management. In *Proc. Int. Conf. on Statistical and Scientific Database Management*, 1998.
2. T. Bultan, R. Gerber, and C. League. Composite model checking: Verification with type-specific symbolic representations. *ACM Transactions on Software Engineering and Methodology*, 9(1):3–50, January 2000.
3. T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.
4. W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. Conf. on Computer Aided Verification*, 2000.
6. C. A. Ellis. Information control nets: A mathematical model of office information flow. In *ACM Proc. Conf. Simulation, Modeling and Measurement of Computer Systems*, pages 225–240, August 1979.
7. M. Garey and D. Johnson. *Computers and Intractability A Guide to the theory of NP-Completeness*. Freeman, 1979.
8. D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–154, April 1995.
9. S. Graf and H. Saïdi. Construction of abstract state graph with PVS. In *Proc. Conf. on Computer Aided Verification*, pages 72–83, 1997.
10. R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative workflows that support easy modification and dynamic browsing. In *Proc. Int. Joint Conf. on Work Activities Coordination and Collaboration*, 1999.
11. N. Krishnakumar and A. Sheth. Managing heterogeneous multi-systems tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2), 1995.
12. R. P. Kurshan. Program verification. *Notices of the AMS*, 47(5):534–545, May 2000.
13. K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.
14. C. Medeiros, G. Vossen, and M. Weske. Wasa: a workflow-based architecture to support scientific database applications. In *Proc. 6th DEXA Conference*, 1995.
15. P. Muth, D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz-Dittrich. Enterprise-wide workflow management based on state and activity charts. In *Proc. NATO Advanced Study Institute on Workflow Management Systems and Interoperability*, 1997.
16. H. Saïdi. Model checking guided abstraction and analysis. In *Proceedings of Statica Analysis Symposium*, Lecture Notes in Computer Science. Springer, 2000.
17. M. Schroeder. Verification of business processes for a correspondence handling center using CCS. In *Proc. European Symp. on Validation and Verification of Knowledge Based Systems and Components*, June 1999.
18. W. M. P. van der Aalst and A. H. M. ter Hofstede. Verification of workflow task structures: A Petri-net-based approach. *Information Systems*, 25(1), 2000.
19. Workflow management coalition. `http://www.aiim.org/wfmc`, 2000.