Formal Verification of E-Services and Workflows

Xiang Fu, Tevfik Bultan, and Jianwen Su

Department of Computer Science
University of California at Santa Barbara, CA 93106, USA,
{fuxiang,bultan,su}@cs.ucsb.edu

Abstract. We study the verification problem for e-service (and workflow) specifications, aiming at efficient techniques for guiding the construction of composite e-services to guarantee desired properties (e.g., deadlock avoidance, bounds on resource usage, response times). Based on previously proposed e-service frameworks such as AZTEC and e-FLow, decision flow language Vortex, and our early work on verifying Vortex specifications using model checking and infinite state verification tools, we introduce a very simple e-service model for our investigation of verification issues. We first show how three different model checking techniques are applied to verification of specifications in simple e-service model, where the number of processes is limited to a predetermined number. We then introduce *pid quantified constraints*, a new symbolic representation that can encode infinite system states, to verify systems with *unbounded* and *dynamic* process instantiations. We think that it is a versatile technique and more suitable for verification of e-service specifications. If this is combined with other techniques such as abstraction and widening, it is possible to solve a large category of interesting verification problems for e-services.

1 Introduction

A profound change caused by the Internet and Web is on the manner many e-commerce, consumer software, and telecommunications applications are provided. Emerging standards (e.g., SOAP, UDDI, WSDL, WFDL) and industrial technology (e.g., IBM's Web services Toolkit, Sun's Open Net Environment and JiniTM Network technology, Microsoft's .Net and Novell's One Net initiatives, HP's e-speak) in e-services has focused on providing pragmatic, working systems so that e-services can effectively interact with each other. Recent research [5, 6, 11, 12, 42, 13, 2, 1, 15] focus on complimentary technologies, for modeling at a more fundamental level both e-services themselves, and frameworks for combining them. An important research issue is to develop efficient techniques for guiding the construction of composite e-services to guarantee desired properties (e.g., deadlock avoidance, bounds on resource usage, response times), and more generally for verifying such properties of composite e-services.

Failure in e-services will have potentially a huge impact. Moreover, as service logic gets more and more complex, the design process becomes complicated and error prone.

For example, a Vortex decision flow specification [31] in practical use may consist of hundreds of variables and thousands lines of code. A simple e-service can consist of many concurrently running processes, such as inventory management, electronic payment, online promotion, and automated customer assistance. Design errors can arise from interleaved access over shared data, synchronization between concurrent running business processes, dynamic change of specifications, and very likely the misunderstanding and misinterpretation by programmers on business logic specifications. Hence an interesting issue here is to develop appropriate tools to aid the design of e-service specifications. The aim of this paper is to investigate and develop general verification techniques for quality design of e-services.

Different from the research efforts[24, 33] to model and analyze performance of workflow systems, our main goal is to verify the correctness of logic inside a workflow specification, e.g. consistency of data, avoidance of unsafe system states, and satisfaction of certain business constraints. The verification problem of workflow specification was studied in several contexts. In [35], model checking was applied to verification of Mentor workflow specifications. More specifically, their focus is on properties over graph structures (rather than execution results). A similar approach was taken using Petri-net based structures in [43, 44]. In [17] Davulcu and et al. used concurrent transactional logic to model workflow systems, and verifying safety properties under certain conditions was proved to be NP complete. Another technique for translating business processes in the process interchange format (PIF) to CCS was developed in [41] 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 one primary concern of the present paper.

In our earlier work [22, 23] on verifying Vortex specifications, we studied two different approaches: (1) approximate a specification with a finite state model (machine), and use model checking tools to verify the properties; (2) model a specification with infinite states and use infinite state verification tools such as the Action Verifier [46, 7]. As we show in [22, 23], new techniques are needed in order to make the verification process practical.

A main difference between e-service models [13, 12, 21] and Vortex is that new processes are dynamically created in response to events that may not be predictable. A focus of this paper is to study verification techniques for such dynamic instantiation of processes. For this purpose, we use a simple e-service model to examine the verification problem and develop techniques under several restricted cases. Note that dynamic instantiation of processes can not be handled by existing verification techniques. Indeed, most model checkers only support verification of programs with bounded number of processes.

We propose to use pid quantified constraints to symbolically represent possibly infinite number of system states and to reason about processes ids using existential quantifiers. We developed the corresponding algorithm to compute PRE (precondition)

operator, which is essential to fixpoint computation in model checking. We illustrate this technique using examples. Similar to those Presburger constraint[36] based infinite state model checking approach[8], this technique suffers from the divergence of fixpoint computations. We expect that if combined with abstraction [26, 40, 4] and widening technique[16], this approach can be much more powerful.

The remainder of the paper is organized as follows. In Section 2 we propose the simple e-service model for verification. In Section 3, we introduce different verification techniques to verify systems with bounded number of processes, and give a short review of temporal logics. We use a Vortex application MIHU as a case study, and compare the performance of BDD based finite state model checking and constraint based infinite state approach. In Section 4 we describe our pid quantified constraints to verify systems with dynamic and unbounded process instantiation. Finally we discuss open problems and future research directions in Section 5.

2 A simple E-Service model

Many E-Service and workflow systems such as AZTEC, e-flow, and Vortex[13, 12, 31] employ a relatively simple specification language and avoid the usage of pitfalling language constructs such as pointers. In addition, the computing power of underlying models are limited (e.g. Vortex restricts the dependency graph to be acyclic). This is due to the fact that the design goal of such systems is to facilitate understanding of non-programmers especially business analysts and managers. On the other hand, in some extent, this situation simplifies the application of model checking to the verification of e-services, because most model checkers at the present can only analyze finite state system and do not handle systems with dynamic memory allocation and process instantiation behaviors.

To facilitate our investigation of verification problems, we introduce a simplified model of e-services. This simplified model has computing power and features of most prevalent workflow systems, while at the same time it is simple enough to allow the application of formal verification. In this simple e-service model, we allow dynamic instantiation of processes, data types with infinite domains, shared global variables among concurrent processes, and flexible interprocess synchronization. Variations of this simple model will be studied in the rest of this paper, and several specialized model checking techniques are presented to take advantage of restrictions posted on these variations.

We now formally define the simple e-service model. A *simple e-service schema* consists of a fixed number of *module schemas*, which can communicate with each other by access over *global variables*. A global variable can be of boolean, enumerate or integer type, where the domain of integer type is infinite. During the execution of an e-service schema, each module schema can be instantiated dynamically multiple and possibly unbounded times. We call these instantiations of module schema *module instances* or simply *processes*. Each e-service schema will have a unique *main* module schema,

whose instantiation serves as the entry point at the beginning of execution. During execution, all processes run in parallel, either asynchronously or synchronously. Note that although modules can be composed synchronously, in our intermediate representations to feed into model checkers, they are always transformed to the asynchronously composed form.

Each module schema can have a fixed number of local variables. Again a local variable can be a boolean, enumerate or integer variable. As we will mention later, if each local variables of every process has a finite domain, counting abstraction can be applied to reason about unbounded number of processes. The computation logic of a module is defined by a list of *transition rules*. Each transition rule is expressed in the form of an if-action statement: `if` *condition* `then` *action*. The meaning of the rule is that if *condition* is satisfied then execute the *action*; otherwise the *action* is automatically blocked. At some moment, there might be more than one transition rules enabled, the semantics is to randomly pick up one to execute.

We limit the *condition* in a transition rule to be boolean expression or linear integer constraints over global variables and local variables. An *action* can either be a conjunction of assignments over variables, or a command to instantiate a new process. Global variables can be accessed by all processes, and local variables can only be accessed by its owner. Note that local variables cannot be accessed by other processes even of the same module schema. This restriction on variable scope is natural.

In Fig. 1 we show a little example to illustrate the syntax of simple E-service model. The example consists of two module schemas main and A. Transition rule $t2$ inside module main instantiate a new process of type A, and initialize its local variable pc to be 0. Transition rule $t1$ inside module schema A increments global variable a by 1, and advances its local variable pc to 1. $t2$ and copies of $t1$ that are owned by multiple instances of A run in parallel. It is obvious that we can always instantiate more than two processes of A, and satisfy the CTL property $EF(a = 2)$(eventually a will reach 2).

Global: **Integer** a=0;
Module A (**Integer** pcInit)
Integer pc=pcInit;
Transition Rules:
t1: **if** pc=0 **then** pc'=1 ∧ a'=a+1;
EndModule
Property: **EF** (a=2)

Module main ()
Transition Rules:
t2: **new** A (0);
EndModule

Fig. 1. Example of dynamic process instantiation

3 Verify systems with bounded number of processes

In this section, we discuss the verification techniques for workflow systems with bounded processes. First we give a brief review of model checking technology and temporal logics. Then we provide three different approaches to verify systems with different features. We apply finite state symbolic model checking to one restricted workflow model, the Vortex workflow[31], where integer domain is finite and dependency graphs is acyclic. These features allow us to develop optimization techniques such as initial constraints projection and variable pruning. It is proved that both finite and infinite approach can converge in finite steps when verifying such systems without loops. Then we relax the restriction on the domain of integer variables, and apply integer constraint based infinite state approach. We show experimental results on a Vortex application MIHU, and compare the performance of the two methods. Finally we discuss the solution to remedy one drawback of the infinite approach. We show that hybrid predicate abstraction technique can not only effectively shrink model size but help fixpoint computation to converge in finite steps as well.

3.1 Model checking

In a landmark paper[37] in 1977, Pneuli argued that *temporal logic* can be very useful to specify correctness of programs, especially the nonterminating reactive systems with concurrent components. With the aid of powerful operators to express concept such as "eventually" and "always", temporal logic wins over Hoare Logic to specify time-vary behaviors. From late 70's thrived many flavors of temporal logics. Debate over which one is preferable is hot, especially for LTL(Linear Temporal Logic)[38] and CTL(Computation Tree Logic)[14]. More detail discussions can be found in [20]. In this paper all specification of properties will be written in CTL and its extensions.

In CTL formulas temporal operators such as X(in next state), F(eventually) and G(globally) must be immediately preceded by a path quantifier A(for all paths) or E(exists a path). For example, mutual exclusion property for a two-processes system can be expressed as $AG\neg(pc1{=}cs \wedge pc2{=}cs)$, and progress property can be expressed in formula $AG((pc1{=}wait \Rightarrow AF(pc1{=}cs) \wedge (pc2{=}wait \Rightarrow AF(pc2{=}cs))$. When processes are instantiated dynamically, we can enhance CTL with quantifiers. For example, the mutual exclusion property and progress property can be expressed in a quantified form as $AG(\forall_{p1\neq p2}\neg(pc[p1]{=}cs \wedge pc[p2]{=}cs))$, and $AG(\forall p(pc[p]{=}wait \Rightarrow AF(proc[p].pc{=}cs))$.

Classified by the representation of system states, there are two types of model checking, explicit state model checking and symbolic model checking. Explicit state model checking[29, 30, 28] has a close relationship with $B\ddot{u}chi$ automata[45], a finite state machine that accepts infinite words. All verification problems in explicit model checking can be transformed to satisfiability test of $B\ddot{u}chi$ automata. In practice we are more inclined to use symbolic model checking[10], as system states are represented more

compactly, much bigger systems can be verified. In symbolic model checking, Binary decision diagram(BDD)[39] is a most frequently used symbolic representation to verify finite state systems, and Presburger formulas[36] is popular in verifying infinite state systems.

Now we give a short introduction on how CTL properties are verified. Suppose that we already have a workflow specification formally modeled. The transition system $T = (S, I, R)$ consists of a *state space* S, a *set of initial states* $I \subseteq S$, and a *transition relation* $R \subseteq S \times S$. Given a set of states p, its pre-condition $\text{PRE}(p, R)$ are all the states that can reach a state in p with a single transition in R (i.e., the set of predecessors of all the states in p), i.e. $\text{PRE}(p, R) = \{s : \exists s' s.t. s' \in p \wedge (s, s') \in R\}$. $\text{POST}(p, R)$ is defined similarly as $\text{POST}(p, R) = \{s : \exists s' s.t. s' \in p \wedge (s', s) \in R\}$. Based upon $\text{PRE}(p, R)$ we are able to verify CTL properties for a transition system $T = (S, I, R)$. For example, let $EF(p, R)$ represents the set of all possible states from which there exists an execution path such that eventually p is satisfied. Then $EF(p, R)$ is computed by least-fixpoint $EF(p, R) = \mu x.(p \vee \text{PRE}(x, R))$. Transition system T satisfies CTL property EFp if and only if initial states $I \subseteq EF(p, R)$. More details about CTL verification can be found in [34].

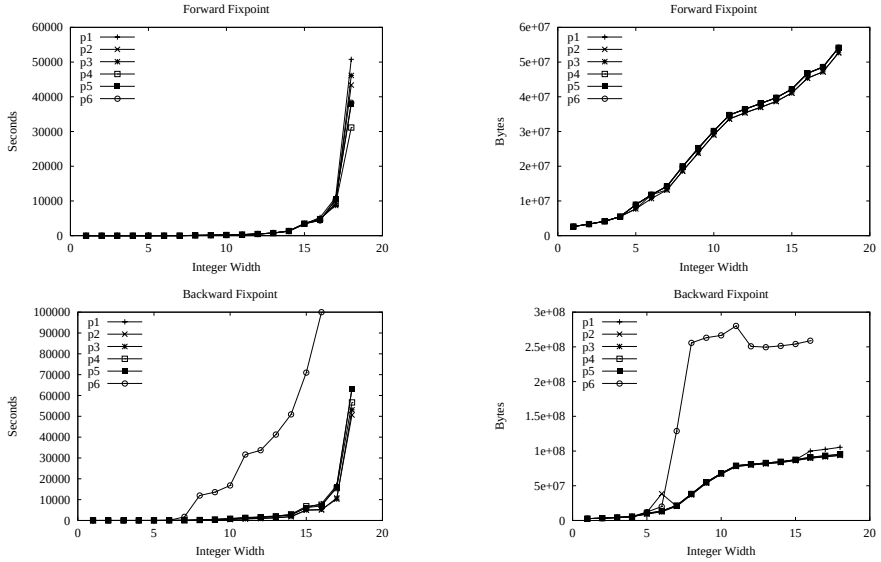
3.2 BDD-based finite approach

In practice many workflow systems can be mapped to a restricted variation of our simple E-service model. For example, Vortex[31] workflow can be regarded as a restricted simple E-service model such that domains of integer variables are finite and dependency graph is acyclic. By taking advantage of these restrictions, we are able to apply BDD-based finite model checking and develop certain optimization techniques. We now demonstrate these optimizations and present experimental results based on our earlier work[22] to model check Vortex workflow[31] with symbolic model checker SMV[34].

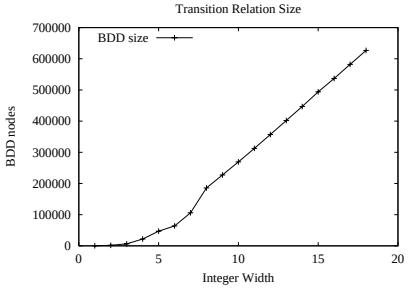
Given a simple E-service schema and all its transition rules, if a variable A is used to compute some variable B, either in the enabling condition part or action part of a transition rule, we say that variable B is dependent on variable A. Following this definition, we can derive a dependency graph for each E-service schema. A simple E-service schema with *acyclic* dependency graph will have many good properties, for example the declarative semantics, i.e. given a fixed input any legal execution sequence will eventually generate the same output. Based on this observation, if the desired property is about values of leaf nodes in the dependency graph, it suffices to check one legal execution path. This idea is similar to partial order reduction[25]. As we only generate an equivalent part of transition system, verification cost is lowered. Moreover, a better structured BDD transition relation can be generated, and the overall BDD transition size is linear on program size. More discussions can be found in [22].

By taking advantage of acyclic dependency, we are able to develop two more optimizations named *variable pruning* and *initial constraints projection*. The idea of variable pruning is based on the observation that in a simple E-service schema with acyclic

dependency graph each variable has a “lifespan”. Outside of its lifespan, a variable becomes no use for execution, and we can assign it a “don't care” value. The assignment, is in fact to eliminate that variable in the BDD representation. Thus during each step of fixpoint computation, the BDD representation encodes only the “active” variables. Hence we can successfully reduce the state space of the problem to the state space over the set of active variables, which is only a small portion of the set of all variables. Similar to variable pruning, source variables(in dependency graph those variables have outgoing edges only) can be “lazily assigned” until they are first referenced. To keep the equivalence to original model, initial constraints are projected to those lazy assignments. This helps to alleviate BDD operations on sorted arrays, more details can be found in [22].



SMV Time Usage SMV Memory Usage



Transition Size

Fig. 2. Experimental results of MIHU

We took a Vortex application MIHU[22] as a case study. MIHU consists of around forty integer variables and hundreds of source lines. We are able to prove all correct properties, and managed to identify violation of two proposed properties on MIHU, which was caused by missing bounds on some of attributes. The graphs shown in Figure 2 show the time used and memory consumed for both backward and forward image computation by SMV. The bottom one shows the size of the BDD that encodes the transition relation.

There are some interesting results reflected in the SMV data. First, it can be observed that the time consumed increases exponentially with the integer width due to the exponential cost associated with image computation. However memory consumption does not increase as sharply demonstrating that BDDs generate a compact encoding of the state space. Second, the transition relation size increases almost linearly with the integer width, which reflects the fact that the BDD encoding of linear integer constraints should have a linear size over the integer width. This is reflected in the graph at the bottom of Figure 2. Third, we observed that to set an appropriate integer width for verification of a property is important for finite state verification. For most of the properties the BDD encoding is not sound if we restrict the integer domains too much. For example for some properties if we restrict the integer width to 4 bits the results that SMV gives are incorrect (i.e., SMV reports true for false properties or false for true properties). This is due to the fact that the constants used in the Vortex schema/property exceed the range of variables and lead to an incorrect modeling. The problem of determining what is the smallest integer width that guarantees the soundness of finite-state verification is an interesting direction for future research.

3.3 Integer constraints based infinite approach

As mentioned in the previous subsection, BDD based finite state approach does not scale well with the integer width. This is due to the fact that BDD symbolic representations are specialized for encoding boolean variables and become inefficient when used to represent integer constraints, which should be represented by more efficient Presburger arithmetic formulas. Infinite-state representations based on linear arithmetic constraints have been used in verification of real-time systems, and infinite-state systems [3, 9, 27] which are not possible to verify using explicit representations. Action Language Verifier [7], based upon Composite Symbolic Library[46] that manipulated both BDD and Presburger package, is such an infinite-state symbolic model checker developed for automated verification of CTL properties of Action Language specifications. Action Language specifications are modular, each module is defined as a composition of its actions and submodules. The similarity of syntax allowed us to take Action verifier as a rapid prototyping tool to investigate the infinite model checking approach.

Action verifier use *composite formulas* to represent transition system $T = (S, I, R)$. A composite formula is obtained by combining boolean and integer formulas with logical connectives. Boolean formulas are represented in the form of ROBDD [34], and in-

teger formulas In Composite Symbolic Library are stored in a disjunctive normal form representation provided by Presburger Arithmetic manipulator Omega Library[32]. In this representation, a Presburger formula is represented as the union of a finite list of polyhedra, .

Let p_i^B and r_i^B be boolean logic formulas, and p_i^I and r_i^I denote Presburger arithmetic formulas. Given a set of states $p = \bigvee_{i=1}^{n_p} p_i^B \wedge p_i^I$ and a transition relation $R = \bigvee_{i=1}^{n_R} r_i^B \wedge r_i^I$, The PRE operator, which is essential for fixpoint computation, can be computed by the following equation, as a result of distribution law of existential quantification.

$$\text{PRE}(p, R) = \bigvee_{i=1}^{n_R} \bigvee_{j=1}^{n_p} \text{PRE}(p_j^B, r_i^B) \wedge \text{PRE}(p_j^I, r_i^I)$$

Note that $\text{PRE}(p_j^B, r_i^B)$ can be computed by existentially eliminating boolean variable using a BDD manipulator [34] and $\text{PRE}(p_j^I, r_i^I)$ can be computed by calling Presburger arithmetic manipulator[9]. Same observation holds for POST function. Based upon PRE operator we can continue to use traditional CTL model checking algorithms.

The translation from Vortex Decision Flows to Action Language is straightforward, and we present the experimental results on MIHU in Figure 3. Comparing with the finite approach, we do not have to worry about the integer width when using the Action Language Verifier, the verification results are provably sound. Other than property 3 Action Language Verifier was able to prove or disprove all the properties. For property 3 the Action Language Verifier did not converge, which demonstrates the high complexity associated with infinite-state model checking. The fifth column in the table shows the smallest integer width when the Action Language Verifier starts to outperform SMV. Hence, even for a finite problem instance, it is better to use an infinite-state model checker rather than a finite-state model checker after these integer widths. The results also show that the Action Language Verifier uses more memory than SMV. Part of the reason could be that the Action Language Verifier uses DNF to store integer constraints which may not be as compact as the BDD representation. As constraint-based tools such as Action Language Verifier are not as mature as SMV which has been studied for more than a decade, we think there is still room for improvement in the performance of constraint-based infinite state model checkers such as Action Language Verifier.

3.4 Hybrid predicate abstraction

There are two basic difficulties in application of Action Language Verifier (or any other infinite-state model checker) to verification of workflows: 1) The large number of variables in a workflow specification can cause the infinite-state symbolic representations such as polyhedra to become prohibitively expensive to manipulate. 2) Since variable domains are not bounded the fixpoint computations may not converge within finite steps. The simple example on the left side of Fig. 4 shows that sometimes even a simple loop can make Action Language Verifier diverge.

Property	Time (Seconds)	Memory (Mb)	Winning Bits against SMV(Backward)	Winning Bits against SMV(Forward)
p1:	303s	17.8	9	12
p2:	271s	17.8	9	11
p3:	diverged			
p4:	271s	17.8	9	11
p5:	158347s	688.3	19	19
p6:	131070s	633.3	17	19

Fig. 3. Verification Results for Action Language Verifier

Global:
Integer y ;
Initial: y=1;
Module main()
Transition Rules:
t1: y'=y+1;
EndModule
Property: **AG**(y≠−1)

Global:
Bool b1,b2 ; // b1: y=−1, b2: y>0
Initial: b1 ∧ $\overline{\text{b2}}$;
Module main()
Transition Rules:
t1: **if** b1∨b2 **then** b1'=$false$;
else b1'=?;

if b2 **then** b2'=$true$;
elseif b1 **then** b2'=$false$;
else b2'=?;
EndModule
Property: **AG**($\overline{\text{b1}}$)

Fig. 4. Example that fixpoint computation can not converge

When Action tries to verify property $AG(y{\neq}{-}1)$, it has to first compute $EF(y{=}{-}1)$, and then check whether the set $EF(y{=}{-}1) \cap \{y{=}1\}$ is an empty set. Since EFp is computed by least-fixpoint $\mu x.(p \vee \text{PRE}(x))$, and $\text{PRE}^n(y{=}{-}1)$ is $\{y = -1{-}n\}$, Action can not converge in a finite number of steps.

By using *predicate abstraction*[40] we can alleviate the problem. The idea is to extract a boolean "abstract" model, and verify properties on this smaller and finite model. Given a list of integer predicates $B_1, ..., B_n$ and an integer program $\mathcal{C}$, by predicate abstraction we can derive an abstract system $\mathcal{A}$ whose system state is a n-tuple $(b_1, ..., b_n)$. In the abstract transition relation, each transition rule is derived from a corresponding transition rule in concrete system, and each abstract transition rule is a conjunction of assignments over abstract boolean variables. As an example, we give the abstract version of the little loop example on the right side of Fig. 4, the abstract program can be successfully verified and thus prove the correctness of the concrete program.

The cost of abstraction is pretty expensive. Suppose that the number of predicates to be abstracted is k, the complexity to compute a single abstract transition rule is $O(k * 3^k)$. For rule based E-Service systems and Workflow systems such as Vortex, there are a lot of switch case statements, thus one integer variable may be involved in many predicates. To abstract such integer variables out proves not successful. We resort to an hybrid approach – just abstract out those variables that are hard to handle by infinite-state verifier, and leave others untouched. We verified a WebShop workflow specification using three approaches: the pure infinite-state fixpoint computation, the pure predicate abstraction and the hybrid approach. The experimental results are displayed in Table. 1. As shown in the table, hybrid approach can achieve a better performance in practice. Currently to identify the set of variables to be abstracted out still needs human guidance, to automatically identify them remains as one of our future research directions. More details can be found at [23].

DataSet	Transition Relation Construct Time	Verification Time	Number of Vars Abstracted Out	Number of Predicates
pure infinite-state fixpoint	0.46s	$>$ 1 hour	0	0
hybrid1	67s	206s	2	5
hybrid2	144s	53s	3	6
pure abstraction	$>$ 1 hour	-	26	57

Table 1. Experimental Results

4 Verify systems with unbounded number of processes

We discuss techniques used to tackle unbounded and dynamic instantiation of processes in this section. It is well known that model checkers can not handle systems with a large number of processes very effectively, unless some other abstraction techniques are applied. We present an existing technique called counting abstraction to handle system with finite yet unbounded processes. We show that this technique has limitations, and then we present a more flexible and versatile framework using "pid quantified constraints".

4.1 Counting abstraction

As the number of processes grows, the cost of model checking can grow exponentially. For example, one can easily verify whether one solution of *producer consumer* problem with two processes satisfies mutual exclusion property, yet it is almost impossible to do

so for 100 processes without any further optimization techniques. By applying *counting abstraction* [18], we are able to prove such problems with unbounded number of processes.

The main idea of counting abstraction is to define a counter for each local state of a module schema, so that the number of processes in this local state can be recorded. By doing so, one can easily verify mutual exclusion problem by checking whether the counter of that critical state will ever exceed 1. For example, in Fig. 1, if we change the data type of local variable pc to enumerate type, we can apply the counting abstraction. Suppose the enumerate type for pc contains two elements loc_0 and loc_1, as pc is the only local variable of module schema A, there are only two local states for any instance of A, naturally we label them as loc_0 and loc_1. Then to define the abstract system, we will declare two integer variables for local state loc_0 and loc_1 respectively, and in transitions we will add operations on these counters to record the status change of processes. For example, in $t1$ we will increment counter for loc_1 and decrement counter for loc_0 by 1.

Counting abstraction has been successfully applied in verifying parameterized cache coherence protocol[18], and Client-Server communication protocols[19]. However there are limitations to apply this technique. Since one has to define a counter for each possible local state of each "process schema", for counting abstraction to work, we need local states of processes to be finite. In another word, processes can not have data types of infinite domain, such as integers. Another drawback is that since local states are totally abstracted away, only some particular properties can be expressed in the abstract system. We can not reason about progress properties like "if process 1's state is wait, then eventually its state can reach critical section", because we only know about the number of processes in some state, but have no information about any specific process. We propose a more versatile framework in the next subsection, which allows processes with infinite local states.

4.2 Pid quantified constraints

We consider dynamic and unbounded process instantiation in this section. The main approach is to utilize a new symbolic representation named *pid quantified constraints* to represent infinite system states and reason about process ids. We first define system state for simple e-service model, and then present the concept of pid quantified constraints. We show how to construct a constant sized intermediate transition relation for systems with dynamic process instantiation. Finally we use a simple example to illustrate the algorithm to compute pre-condition operator $\text{PRE}(p, R)$.

System state A *system state schema* is a tuple $(\mathbf{G}, \mathbf{P}, \mathbf{L})$, and *system state* its valuation. In a system state schema $(\mathbf{G}, \mathbf{P}, \mathbf{L})$, $\mathbf{G}$ is the set of all global variables, $\mathbf{P}$ consists of the instantiation counters for each module schema, and $\mathbf{L}$ is the set of local variables of all processes(include those inactive processes). $\mathbf{L}$ can be regarded as

a list of unbounded arrays. In $\mathbf{L}$ for the nth instantiation of a module schema A, we use $A.var_i[n]$ to denote its local variable var_i, and we call n the *index*. In the unbounded array, elements of those inactive processes are assigned uninitialized value $\perp$, i.e. $i{>}A.Cnt \Rightarrow A.v[i]{=}\perp$. For example, for the simple program listed in Fig. 1 its system state schema is $(a, A.Cnt, A.pc[])$, and one possible state is $(2, 2, [1, 1, \perp, \perp, \ldots])$. In this state, process A has been instantiated twice.

Pid quantified constraints We now define the concept of *pid quantified constraint*, which can be used to encode infinite many system states. Let *axiom* be a boolean variable or linear integer constraint over global variables and local variables, and *expression* a boolean expression over axioms. One *pid quantified constraint* $\overset{A}{\exists}_{\boldsymbol{a}} \ldots \overset{L}{\exists}_{\boldsymbol{l}} expr$ is an $expression$ existentially quantified by a list of *unique existential quantifiers*. In a pid quantified constraint only bounded variables can be used to index local variables, and bounded variables are only used as index variables. For example, formula $\overset{A}{\exists}_{a_1,a_2} a_1{<}a_2 \wedge A.pc[a_1]{=}0$ and formula $\overset{A}{\exists}_{a_1} A.pc[a_1]{=}A.pc[x]$ are not pid quantified constraints. The meaning of *unique existential quantifier* $\overset{A}{\exists}_{\boldsymbol{a}}$ is a little different from that of existential quantifiers in first order logic. We require that all variables in the set $\boldsymbol{a}$ are only used to index local variables of Module A, and each index variable should be mapped to a unique number no greater than $A.Cnt$, i.e. a state s satisfies a pid quantified formula $\overset{A}{\exists}_{a_1,\ldots a_n} expr$ if and only if there exist a valuation v of $a_1, \ldots, a_n$ such that the state s satisfies $expr$ and $\forall_{a_i \neq a_j} v(a_i){\neq}v(a_j) \wedge v(a_i){\leqslant}A.Cnt$. For example, state $(2, 2, [1, 1, \perp, \ldots])$ satisfies formula $\overset{A}{\exists}_{a_1,a_2} A.pc[a_1]{=}1 \wedge A.pc[a_2]$, but it does not satisfies formula $\overset{A}{\exists}_{a_1,a_2,a_3} A.pc[a_1]{=}1 \wedge A.pc[a_2]{=}1 \wedge A.pc[a_3]{=}1$.

In the following context, we usually use a pid quantified constraint f to refer to the set of states that satisfy formula f. For example, $\overset{A}{\exists}_{i,j} A.pc[i] \leqslant A.pc[j]$ represents all possible states such that there are at least two processes of module schema A, and the local variable pc of one process is less than or equal to the other. Note that in fact $\overset{A}{\exists}_{i,j} A.pc[i]{\leqslant}A.pc[j] \equiv \overset{A}{\exists}_{i,j} true$, and it might be amazing that $\overset{A}{\exists}_{i,j} A.pc[i]{+}A.pc[j]{\neq}5 \equiv \overset{A}{\exists}_{i,j} true$. As far as we know, it is not clear the subset test of two pid quantified constraints is decidable, however there exist conservative algorithms (sufficient condition) to do subset test. The idea of the conservative algorithm is to generate two Presburger formulas from the pid quantified constraints, compare them, and use the result to tell the comparison between the original pid quantified constraints. The process to generate Presburger formulas needs to consider all permutations of index variables.

Satisfiability of a pid quantified constraint is decidable, as the problem can be transformed into the satisfiability problem of Presburger formulas. An pid quantified constraint $\overset{A}{\exists}_{\boldsymbol{a}} \ldots \overset{L}{\exists}_{\boldsymbol{l}} expr$ is satisfiable if and only if $expr[A.v[a]/A_{v_a}]$ is satisfiable. Here we replace every appearance of array elements with a free integer variable in $expr$, as

shown by $A.v[a]/A_{v_a}$. For example, $\overset{A}{\exists}_{i,j}(A.pc[i]{<}A.pc[j] \wedge A.pc[j]{<}A.pc[i])$ is not satisfiable because Presburger formula $A_{pc_i}{<}A_{pc_j} \wedge A_{pc_j}{<}A_{pc_i}$ is not satisfiable.

Transition relation In traditional model checking transition relation size grows in proportion to the number of instantiations of each module schemas. This is because each instantiation has his own copy of transition rules. For example, in traditional model checking, if there are two instances of module schema A in Fig. 1, we will have two copies of transition rule $t1$ asynchronously composed in the transition relation, and the two copies have only a slight difference in accessing the local variable pc. Now with the power of quantifiers, and given that local variables are stored using unbounded arrays, we can make the representation of transition relations much more succinct, and moreover its size is always constant no matter how many instances there are. For example, our first order representation of $t1$ and $t2$ in Figure 1 are listed in the Equation 1 and 2. The semantics of $t1$ is "if there exist a process of module schema A whose local variable pc is 0, then we advance its pc to 1 and increment the global variable a by 1, and for all other variables we let them keep their original values". In Equation 2, the semantics is to increment counter of schema A by 1 and initialize the new local variable with 0, and then let all other variables keep their original values.

$$t1{:}\ \overset{A}{\exists}_i A.pc[i]{=}0 \wedge A.pc[i]'{=}1 \wedge a'{=}a{+}1 \wedge \forall^A_{j\neq i} A.pc[j]'{=}A.pc[j] \wedge A.Cnt'{=}A.Cnt \quad (1)$$

$$t2{:}\ A.Cnt'{=}A.Cnt{+}1 \wedge A.pc[A.Cnt']'{=}0 \wedge a'{=}a \wedge \forall^A_{i\neq A.Cnt'}(A.pc[i]'{=}A.pc[i]) \quad (2)$$

Verification As discussed in previous sections, pre-condition operator PRE needs to be defined before doing fixpoint computation to model check CTL properties. Without loss of generation, we can assume that all transition actions can be divided into two types, pure assignment action and process instantiation action. We now discuss how to compute PRE for these two types of actions.

Type I. Let tuple $(\mathbf{G}, \mathbf{P}, \mathbf{L})$ be the system schema, a type I transition rule τ_A of of Module A can expressed in the following form.

$$\tau_A := \overset{A}{\exists}_i \Big(\mathcal{T}(I^G, O^G, I^L_i, O^L_i) \wedge \textsc{same}(\hat{O}) \Big) \quad (3)$$
$$\textbf{where } \textsc{same}(V) := \bigwedge_{v\in V} v' = v$$

In Equation 3 index variable i identifies the process to take action, set I^G, O^G, I^L_i, O^L_i represents the global input, output, local input and output variables respectively, obviously I^L_i and O^L_i contains local variables indexed by i only. $\hat{O} = \mathbf{L} \cup \mathbf{G} \cup \mathbf{P} - O^G - O^L_i$ represents the set of variables that should preserve their original values. For example, for the transition rule $t1$ shown in Equation 1, $I^G{=}O^G{=}\{a\}$, $I^L_i = O^L_i = \{A.pc[i]\}$, and $\hat{O} = \{A.pc[j] \mid j \neq i\}$.

Suppose a pid quantified constraint $\overset{A}{\exists}_{\boldsymbol{a}} \ldots \overset{L}{\exists}_{\boldsymbol{l}} expr(L_s^A, L_s^B, ..., L_s^L, G_s)$ is used to represent the current states set S. Here G_s is the set of global variables appeared in $expr$, L_s^A, ..., L_s^L are the sets of local variables of module A to L that appeared in $expr$ respectively. The *next form* of S is to simply substitute all variables appeared in $expr$ with their "primed" forms. For example, the next form of $\overset{A}{\exists}_{a_i} A.pc[a_i]{=}1$ is $\overset{A}{\exists}_{a_i} A.pc[a_i]'{=}1$.

Then PRE(S,τ_A) can be computed in two steps. First we compute the conjunction $\mathcal{C} = \mathcal{S}' \wedge \tau_A$, and then we do existential quantification on $\mathcal{C}$ to eliminate all "primed" variables, i.e. let $\mathcal{X}$ be the set of "primed" variables appeared in $\mathcal{C}$. $\text{PRE}(S, \tau_A) := \exists_{\mathcal{X}} \mathcal{C}$. The existential quantification process can be simply accomplished by Presburger solver and BDD manipulator. Now the formula to compute $\mathcal{C}$ is listed as follows.

$$
\begin{aligned}
\mathcal{C} \quad := \quad & \overset{A}{\exists}_{\{i\}\cup\boldsymbol{a}} \ldots \overset{L}{\exists}_{\boldsymbol{l}} expr' \wedge \mathcal{T}(I^G, O^G, I_i^L, O_i^L) \wedge \text{SAME}_1(V_1) \wedge \text{SAME}_2(V_2) \\
& \vee \bigvee_{a_j \in \boldsymbol{a}} \overset{A}{\exists}_{\boldsymbol{a}} \ldots \overset{L}{\exists}_{\boldsymbol{l}} expr' \wedge \mathcal{T}(I^G, O^G, I_{a_j}^L, O_{a_j}^L) \wedge \text{SAME}_1(V_1') \wedge \text{SAME}_2(V_2') \\
\textbf{where } & V_1 = L_s^A \cup \ldots \cup L_s^L \cup G_s - O_i^L \cup O^G \\
& V_2 = \mathbf{G} \cup \mathbf{L} \cup \mathbf{P} - L_s^A \cup \ldots \cup L_s^L \cup G_s \cup O_i^L \cup O^G \\
& V_1' = L_s^A \cup \ldots \cup L_s^L \cup G_s - O_{a_j}^L \cup O^G \\
& V_2' = \mathbf{G} \cup \mathbf{L} \cup \mathbf{P} - L_s^A \cup \ldots \cup L_s^L \cup G_s \cup O_{a_j}^L \cup O^G
\end{aligned} \tag{4}
$$

As shown in Equation 4, $\mathcal{C}$ is a disjunction of $|\boldsymbol{a}| + 1$ clauses. The first clause (started with quantifiers $\overset{A}{\exists}_{\{i\}\cup\boldsymbol{a}}$) handles the case when none of index $a_j{\in}\boldsymbol{a}$ is mapped to a number equal to i, and the other $|\boldsymbol{a}|$ clauses handle the cases when there is exactly one $a_j{\in}\boldsymbol{a}$ equal to i. Note that due to the definition of "unique existential quantifier" we only have to consider these $|\boldsymbol{a}| + 1$ cases. In the last $|\boldsymbol{a}|$ clauses, since a_j is identical to i in τ_A, we substitute all appearance of i in $\mathcal{T}$ with a_j, and this operation is represented by set $I_{a_j}^L$ and $O_{a_j}^L$ inside function $\mathcal{T}$ in the last $|\boldsymbol{a}|$ cases. Now the last problem is how to deal with SAME. We split $\text{SAME}(\hat{O})$ into two parts $\text{SAME}_1(V_1)$ and $\text{SAME}_2(V_2)$. It is clear that $\hat{O} = V_1 \cup V_2$, and V_1 is a finite set while V_2 is an infinite one. The most important fact is that in the formula of $\mathcal{C}$ each variable $v{\in}V_2$ appears in the subformula of SAME_2 only, with the form $v' = v$. Thus after existential quantification of v' in the second step, v will not appear in the result. This guarantees the finite length of $\text{PRE}(S, \tau_a)$, because subformula $expr$, $\mathcal{T}$ and SAME_1 all have finite length.

Example 1. Let the current state S represented by formula $a{=}1 \wedge \overset{A}{\exists}_{a_1} A.pc[a_1]{=}0$, and the transition τ_A be the $t1$ shown in Equation 1. Then the next state S' is $a'{=}1 \wedge \overset{A}{\exists}_{a_1} A.pc[a_1]'{=}0$, and the conjunction $\mathcal{C}(\mathcal{S}', \tau_{\mathcal{A}})$, according to Equation 4, is computed as follows. Note that in the second clause(started with quantifier $\overset{A}{\exists}_{a_1}$) because $A.pc[a_1]'$

is assigned 0 and 1 in $expr'$ and $\mathcal{T}$ at the same time, the second clause is not satisfiable, and hence represents empty set.

$$
\begin{aligned}
\mathcal{C} \quad := \quad & \overset{A}{\exists}_{i,a_1}(a'{=}1 \wedge A.pc[a_1]'{=}0) \wedge (A.pc[i]{=}0 \wedge A.pc[i]'{=}1 \wedge a'{=}a{+}1) \wedge \\
& \qquad \textsc{same}_1(V_1) \wedge \textsc{same}_2(V_2) \\
& \vee \overset{A}{\exists}_{a_1} \Big(a'{=}1 \wedge A.pc[a_1]'{=}0) \wedge (A.pc[a_1]{=}0 \wedge A.pc[a_1]'{=}1 \wedge a'{=}a{+}1) \wedge \\
& \qquad \textsc{same}_1(V_1') \wedge \textsc{same}_2(V_2') \\
\textbf{where} \ & V_1{=}\{A.pc[a_1]\}, \ \ V_2{=}\{A.pc[x] \mid x{\neq}a_1 \wedge x{\neq}i\} \cup \{A.Cnt\} \\
& V_1'{=}\phi, \ \ V_2'{=}\{A.pc[x] \mid x{\neq}a_1\} \cup \{A.Cnt\} \\
:= \quad & (a{+}1{=}a'{=}1) \wedge \overset{A}{\exists}_{i,a_1}(A.pc[a_1]'{=}0 \wedge A.pc[i]{=}0 \wedge A.pc[i]'{=}1 \wedge A.pc[a_1]'{=}A.pc[a_1])
\end{aligned}
$$

Thus, after existential quantification, we get

$$\textsc{pre}(S, \tau_A) := \quad a = 0 \wedge (\overset{A}{\exists}_{i,a_1} A.pc[i] = 0 \wedge A.pc[a_1] = 0)$$

Type II We adopt a similar methodology to handle the "process instantiation actions". This time we analyze the relationship between index variables and the instance counter, as shown in the following example.

Example 2. Suppose S is represented by formula $a{=}1 \wedge \overset{A}{\exists}_{a_1} A.pc[a_1]{=}0$, and the transition $\tau 2_A$ is $t2$ shown in Equation 2. Then conjunction $\mathcal{C}{=}\mathcal{S}' \wedge \tau 2_A$ is computed as follows.

$$
\begin{aligned}
\mathcal{C} := \ & (a'{=}1 \wedge A.pc[A.Cnt']'{=}0) \wedge (A.Cnt'{=}A.Cnt{+}1 \wedge A.pc[A.Cnt']{=}0) \wedge \\
& \qquad \textsc{same}_1(V_1) \wedge \textsc{same}_2(V_2) \\
& \vee (a'{=}1 \wedge \overset{A}{\exists}_i (A.pc[i]'{=}0) \wedge (A.Cnt'{=}A.Cnt{+}1 \wedge A.pc[A.Cnt']'{=}0) \wedge \\
& \qquad \textsc{same}_1(V_1') \wedge \textsc{same}_2(V_2') \\
\textbf{where} \quad & V_1{=}\{a\}, \ V_2{=}\{A.pc[x] \mid x{\neq}A.Cnt'\} \\
& V_1'{=}\{a, A.pc[i]\}, \ V_2'{=}\{A.pc[x] \mid x{\neq}A.Cnt' \wedge x{\neq}i\} \\
:= \ & a'{=}a{=}1 \wedge A.Cnt'{=}A.Cnt + 1 \wedge A.pc[A.Cnt']{=}0
\end{aligned}
$$

As shown in the calculation steps above, we have two disjuncted clauses in $\mathcal{C}$. The first clause handles the case when index variable i is instantiated as $A.Cnt'$ (we replace all appearance of i in S' and take off the quantifier on i). The second one handles the case when i and $A.Cnt'$ are mapped to two different numbers. It is easy to see that the second clause is a subset of the first clause, and we can eliminate it in the simplified form. Finally, we do existential elimination, and get the result $\textsc{pre}(S, \tau 2_A) \equiv a = 1$.

Using pid quantified constraints, we can verify the property $EF(a = 2)$ for the example listed in Fig. 1. As the property EFp is evaluated by fixpoint $\mu x.p \vee \textsc{pre}(x, \tau)$.

We listed each step of the fixpoint computation in Fig. 5. We use an early detection algorithm to check whether the set of initial states is a subset of these intermediate results, so verification converges in five steps.

EF0	$a{=}2$
EF1	$a{=}\ 2 \vee a{=}1 \wedge \overset{A}{\exists}_i A.pc[i]{=}0$
EF2	$a{=}2 \vee a{=}1 \vee a{=}0 \wedge \overset{A}{\exists}_{i,j} A.pc[i]{=}0 \wedge A.pc[j]{=}0$
EF3	$a{=}2 \vee a{=}1 \vee a{=}0 \wedge \overset{A}{\exists}_i A.pc[i]{=}0 \vee a{=}{-}1 \wedge \overset{A}{\exists}_{i,j,k} A.pc[i]{=}0 \wedge A.pc[j]{=}0 \wedge A.pc[k]{=}0$
EF4	$a{=}2 \vee a{=}1 \vee a{=}0 \vee a{=}{-}1 \wedge \overset{A}{\exists}_{i,j} A.pc[i]{=}0 \wedge A.pc[j]{=}0 \vee$ $a{=}{-}2 \wedge \overset{A}{\exists}_{i,j,k,l} A.pc[i]{=}0 \wedge A.pc[j]{=}0 \wedge A.pc[k]{=}0 \wedge A.pc[l]{=}0$

Fig. 5. Evaluation Steps

5 Open problems

Similar to the infinite state approach we discussed before, our verification based on pid quantified constraints suffered from the problem that fixpoint computation might not converge in finite steps. For example, the EF computation for the program in Figure 1 will not converge if we do not use early detection algorithm. One promising remedy we think is to use predicate abstraction to derive a finite abstraction and avoid infinite fixpoint computation. As the semantics of transition relations have been enriched by pid quantifiers, we have to redefine the abstraction algorithm, and this will be one of our future research direction.

Another interesting topic is what type of properties can be verified using pid quantified approach. When initial states are represented by pid quantified constraints, safety property $AG(\forall_{p_1,p_2} f(p_1,p_2))$ can be verified, because $EF(\exists_{p_1,p_2} \overline{f(p_1,p_2)})$ can be computed and encoded by pid quantified constraints, and the satisfiability of its intersection with initial states is decidable. However, when temporal operators and pid quantifiers are mixed, things becomes complex. For example, can the progress property $AG(\forall_{pid} pc[pid]{=}\ \textbf{wait} \Rightarrow AF(pc[pid]{=}\ \textbf{Enter}))$ be verified?

We believe that research effort is still needed on subset test algorithms in practice, as such operations are vital to model check liveness properties. It is also interesting to investigate the simplification of pid quantified constraints.

References

1. S. Abiteboul, V. Aguilera, S. Ailleret, B. Amann, F. Arambarri, S. Cluet, G. Cobena, G. Corona, G. Ferran, A. Galland, M. Hascoet, C.-C. Kanne, B. Koechlin, D. LeNiniven,

A. Marian, L. Mignet, G. Moerkotte, B. Nguyen, M. Preda, M.-C. Rousset, M. Sebag, J.-P. Sirot, P. Veltri, D. Vodislav, F. Watezand, and T. Westmann. A dynamic warehouse for XML data of the Web. *IEEE Data Engineering Bulletin*, 2001.

2. S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proc. ACM Symp. on Principles of Database Systems*, 1998.
3. R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.
4. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI 2001*, 2001.
5. B. Benatallah, B. Medjahed, A. Bouguettaya, A. Elmagarmid, and J. Beard. Self-coordinated and self-traced composite services with dynamic provider selection. Technical report, University of New South Wales, Mar. 2001. (Available at http://sky.fit.qut.edu.au/ dumas/selfserv.ps.gz).
6. R. Breite, P. Walden, and H. Vanharanta. C-commerce virtuality - will it work in the Internet? In *Proc. of International Conf on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2000)*, 2000. (http://www.ssgrr.it/en/ssgrr2000/proceedings.htm).
7. T. Bultan. Action language: A specification language for model checking reactive systems. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 335–344, June 2000.
8. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.
9. 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.
10. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.
11. F. Casati, S. Sayal, and M. Shan. Developing e-services for composing e-services. In *Proceedings of CAISE 2001*, Interlaken, Switzerland, June 2001.
12. F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–163, 2001.
13. V. Christophides, R. Hull, G. Karvounarakis, A. Kumar, G. Tong, and M. Xiong. Beyond discrete e-services: Composing session-oriented services in telecommunications. In *Proc. of Workshop on Technologies for E-Services (TES)*, Rome, Italy, Sept. 2001.
14. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1981.
15. G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *Proc. Int. Conf. on Data Engineering*, 2002.
16. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, Leuven, Belgium, 1992. LNCS 631, Springer-Verlag.

17. H. Davulcu, M. Kifer, C.R.Ramakrishnan, and I.V.Ramakrishnan. Logic based modeling and analysis. In *pods*, 1998.
18. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer-Verlag, 2000.
19. G. Delzanno and T. Bultan. Constraint-based verification of client-server protocols. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, 2001.
20. E. A. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.
21. M. C. Fauvet, M. Dumas, B. Benatallah, and H. Y. Paik. Peer-to-peer traced execution of composite services. In *Proc. of Workshop on Technologies for E-Services (TES)*, Rome, Italy, Sept. 2001.
22. X. Fu, T. Bultan, R. Hull, and J. Su. Verification of vortex workflows. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, April 2001.
23. X. Fu, T. Bultan, and J. Su. Hybrid predicate abstraction for verification of workflow and decision flow systems. Technical report, Computer Science Department, UCSB, January 2002.
24. M. Gillmann, J. Weissenfels, G. Weikum, and A. Kraiss. Performance and availability assessment for the configuration of distributed workflow management systems.
25. P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proceedings of the 2nd Workshop on Computer Aided Verification*, LNCS 697, pages 438 – 449, 1993.
26. S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, June 1997.
27. N. Halbwachs, P. Raymond, and Y. Proy. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *Proceedings of International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1994.
28. G. J. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314, Warsaw, Poland, 1995. Chapman Hall.
29. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
30. G. J. Holzmann and et al. On nested depth first search. In *Proc. of the Second Spin Workshop*, pages 385–389, 1996.
31. 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.
32. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995.
33. A. Kraiss and F. Sch˙

34. K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.
35. 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.
36. D. C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16:323–332, 1978.
37. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–67, 1977.
38. A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, pages 45 – 60, 1981.
39. R.E. Bryant. Graph-based algorithms for boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1986.
40. H. Saidi. Model checking guided abstraction and analysis. In *Proceedings of Statica Analysis Symposium*, Lecture Notes in Computer Science. Springer, 2000.
41. 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.
42. Simple object access protocol (soap) 1.1. W3C Note 08, May 2000. (http://www.w3.org/TR/SOAP/).
43. W. M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, systems and computers*, 8(1):21–26, 1998.
44. 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.
45. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of First IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.
46. T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. Composite symbolic library. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 335–344. Springer-Verlag, April 2001.