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 e-service frameworks such as AZTEC and e-FLow, decision flow language Vortex, 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 when the number of processes is limited to a predetermined number. We then introduce *pid quantified constraint*, a new symbolic representation that can encode infinite many 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

Failure in e-services will have potentially a huge impact. As even a simple e-service can consist of many concurrently running processes (e.g. inventory management, electronic payment and online promotion), design process of e-services becomes more and more complicated. Design errors can arise from interleaved access over shared data, synchronization between 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.

Unlike the research effort [14, 20] to analyze the performance model of a workflow system, our main goal here is to verify the logic correctness of 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 [22], model checking was applied to 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 [29,

30]. In [8] Davulcu and et al. used concurrent transactional logic to model workflow systems, and verifying safety properties under certain environment was proved to be NP complete. Another technique for translating business processes in the process interchange format (PIF) to CCS was developed in [28] 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 [12, 13] on verifying Vortex specifications, we studied two different approaches: (1) approximate a specification with a finite state model (machine), and use symbolic model checker (SMV) to verify the properties; (2) model a specification with infinite states and use infinite state verification tools such as the Action Verifier [31, 2]. As we show in [12, 13], new techniques are needed in order to make the verification process practical.

A main difference between e-service models [6, 5, 11] and decision flow language Vortex [18] 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 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. 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.

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

To facilitate our investigation of verification problems, we introduce a simplified model of e-services. This simplified model captures features of most prevalent workflow systems, while at the same time it is simple enough for formal verification. In this 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 various

model checking techniques are presented to take advantage of the features of 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 accessing *global variables*. A global variable can be of boolean, enumerate or integer type, and the domain of integer type is infinite. During the execution of an e-service schema, a module schema can be instantiated dynamically multiple and possibly unbounded times. We call these instantiations *module instances* or simply *processes*. 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 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. An *action* can either be a conjunction of assignments over variables, or a command to instantiate a new process. We limit the expression appeared in *action* and *condition* to be linear, due to the limitation of our model checkers. Global variables can be accessed by all processes, and local variables can only be accessed by its owner.

Fig. 1 shows a little example of the simple E-service model. There are 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 $t1$ (owned by 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 $\mathbf{EF}(\mathsf{a} = 2)$ (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

We discuss the verification of workflow systems with bounded processes in this section. We start with a brief review of model checking technology and temporal logics, which is the basis for our discussion. Then we present three different approaches (finite state, infinite state model checking, and predicate abstraction) to verify a workflow specification. We show several optimization techniques we have developed, by taking advantage of system features. We also compare the pros and cons of the approaches we presented.

3.1 Model checking

In a landmark paper [24] Pnueli argued that *temporal logic* is very useful for specifying correctness of programs especially reactive systems. With powerful operators to express concepts such as "eventually" and "always", temporal logic wins over Hoare Logic in specifying time-vary behaviors. From late 70's thrived many flavors of temporal logic, for example, LTL (Linear Temporal Logic) [25] and CTL (Computation Tree Logic) [7]. In the rest of this paper, we use CTL and its extensions to specify desired properties of workflow programs.

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, for a two-process system mutual exclusion property is expressed as $\mathbf{AG}\neg(\mathsf{pc}_1{=}cs \ \wedge \ \mathsf{pc}_2{=}cs)$, and progress property is expressed as $\mathbf{AG}(\mathsf{pc}_1{=}wait \ \Rightarrow \ \mathbf{AF}(\mathsf{pc}_1{=}cs)) \ \wedge \ \mathbf{AG}(\mathsf{pc}_2{=}wait \ \Rightarrow \ \mathbf{AF}(\mathsf{pc}_2{=}cs))$. When the number of processes is not predetermined, we can enhance CTL with quantifiers, e.g., mutual exclusion property are expressed as $\mathbf{AG}(\forall_{\mathsf{p}_1\neq\mathsf{p}_2}\neg(\mathsf{pc}[\mathsf{p}_1]{=}cs \ \wedge \ \mathsf{pc}[\mathsf{p}_2]{=}cs))$.

There are two types of model checking techniques, explicit state model checking [17] and symbolic model checking [4]. In practice we are more interested in using symbolic model checking, as system states are represented more compactly, and hence much bigger systems can be verified. To encode system state, for finite state system BDD (Binary decision diagram) [26] is the most popular form; and for infinite state system, we use Presburger formulas [23].

We now give a short review of CTL verification algorithm. Suppose that a workflow program is formally modeled as a transition system $T = (S, I, R)$, where S is the *state space* , $I \subseteq S$ is the set of *initial states*, and $R \subseteq S \times S$ is the *transition relation*. Given a set of states p, the *pre-condition* $\text{PRE}(p, R)$ represents the set of states that can reach p with a single transition in R, i.e. $\text{PRE}(p, R) = \{s : \exists s' \ s.t. \ s' \in p \wedge (s, s') \in R\}$. PRE operator is very important for CTL verification, as all verification problems are finally transformed to the fixpoint computation on PRE. For example, to verify whether system T satisfies CTL property $\mathbf{EF}p$, it is equivalent to check whether $I \subseteq \mathsf{EF}(p, R)$. Here $\mathsf{EF}(p, R)$ represents the set of states that can eventually reach p, and it is computed using least-fixpoint $\mathsf{EF}(p, R) = \mu x.(p \vee \text{PRE}(x, R))$, where PRE operator has to be defined. More details about CTL verification can be found in [21].

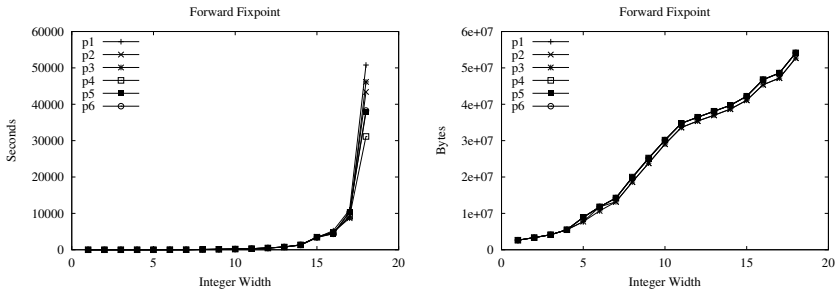
3.2 BDD-based finite approach

In practice many workflow systems can be mapped to a restricted variation of our simple E-service model. For example, Vortex [18] workflow can be regarded as a variation whose integer domain is finite and whose variable 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 [12] to model check Vortex workflow [18] using symbolic model checker SMV [21].

Given a simple E-service schema, if a variable A is used to compute some variable B, we say that B is dependent on A. Following this definition, each E-service schema has a a dependency graph. If this dependency graph is *acyclic*, it is easy to infer that the E-service schema has *declarative semantics*, i.e., given the same 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 only one legal execution path. This idea is similar to partial order reduction [15], as only an equivalent part of transition system is generated, verification cost is greatly lowered.

By taking advantage of acyclic dependency graph, we are able to develop two more optimizations named *variable pruning* and *initial constraints projection*. The idea of variable pruning is that in a E-service schema with acyclic dependency graph, each variable has a "lifespan". Outside of this lifespan the variable is of no use for execution, and hence 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, and we have successfully reduced the state space. 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 [12].

We took a Vortex application MIHU [12] 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 violations of two proposed properties, which was caused by missing bounds on some variables. The graphs shown in Figure 2 demonstrates the time used and memory consumed for forward fixpoint computation by SMV. There are some interesting results reflected in the SMV data. First, the time consumed increases exponentially with the integer width. However memory consumption does not increase as sharply demonstrating that BDDs generate a compact encoding of the state space. Second, we observed that to set an appropriate integer width is important for finite state verification. We can not restrict the integer domains too much, as intermediate results generated during execution can possibly exceed the range of variables and lead to an incorrect modeling. The problem of determining what is the smallest integer width



SMV Time Usage SMV Memory Usage

Fig. 2. Experimental results of MIHU

that guarantees the soundness of finite-state verification is an interesting direction for future research.

3.3 Integer constraints based infinite approach

As shown in the previous subsection, BDD based 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. In stead we can use infinite-state representations based on linear arithmetic constraints [1, 3, 16] to solve this problem. Action Language Verifier [2], based upon Composite Symbolic Library [31] that manipulates both BDD and Presburger package, is such an infinite-state symbolic model checker. 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 uses *composite formulas* to represent transition system. A composite formula is obtained by combining boolean and integer formulas with logical connectives. Boolean formulas are represented in the form of ROBDD [21], and integer formulas in Composite Symbolic Library are stored in a disjunctive normal form representation provided by Presburger Arithmetic manipulator Omega Library [19]. In this representation, a Presburger formula is represented as the union of a finite list of polyhedra. As a result of distribution law of existential quantification, The PRE operator is computed by calling BDD manipulator and Presburger arithmetic manipulator. More details can be found in [31].

The translation from simple E-service model 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.

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

3.4 Hybrid predicate abstraction

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

By using *predicate abstraction* [27] we can alleviate the problem. The idea is to extract a boolean "abstract" model, and verify properties on this smaller 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. This abstract program can be successfully verified and thus prove the correctness of the concrete program.

The cost of abstraction is pretty expensive. Let k be the number of predicates, the complexity to compute a single abstract transition rule is $O(k * 3^k)$ [27]. For rule based E-Service systems, to abstract out all integer variables proves not successful, as there are

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: $\overline{\text{b1}} \wedge$ 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

too many switch case statements and integer predicates. We resort to a hybrid approach – just partially abstract the part causing the divergence of Action, and leave others intact. We verified a WebShop workflow specification using this hybrid approach, and experimental result in [13] shows that it is much more versatile in practice.

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

The main idea of counting abstraction [9] is to define a counter for each local state of a module schema, to record and track the number of processes in this local state. By doing so, one can easily verify mutual exclusion property 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, which contains two elements loc_0 and loc_1, then for module schema A there are only two local states. To apply counting abstraction, we can declare two integer variables as counters for these two local states, and in $t1$ we add operations that increments counter for loc_1 and decrements counter for loc_0 by 1.

Counting abstraction has been successfully applied in verifying parameterized cache coherence protocol [9], and Client-Server communication protocols [10]. However this

technique has certain limitations. Since one has to define a counter for each possible local state, the number of local states of each module schema needs to be finite. In another word, processes can not have data types of infinite domain. 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 it can reach critical section", because we 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

Pid quantified constraint uses a special existential quantifier to reason about process ids, and it can be used as symbolic representation to encode possibly infinite many system states. In this subsection we first define system state, based on which we derive the concept of pid quantified constraints. Then 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 how to compute pre-condition operator $\text{PRE}(p, R)$, which is essential for CTL verification.

System state A *system state schema* is a tuple $(\mathbf{G}, \mathbf{P}, \mathbf{L})$, and a *system state* is its valuation. Here $\mathbf{G}$ is the set of all global variables, $\mathbf{P}$ is the set of module instantiation counters, and $\mathbf{L}$ is a list of unbounded arrays that record local variables of all processes. For example, for the simple program in Fig. 1 its system state schema is $(\mathsf{a}, \mathsf{A.Cnt}, \mathsf{A.pc}[])$, and one possible state is $(2, 2, [1, 1, \perp, \perp, \ldots])$, where $\perp$ represents the "uninitialized value". In this state, module schema A has been instantiated twice.

Pid quantified constraints A *pid quantified constraint* $\overset{A}{\exists}_{\boldsymbol{a}} \ldots \overset{L}{\exists}_{\boldsymbol{l}}\ expr$ is a quantifier-free expression $expr$ (linear constraints connected by boolean operators) 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 $\overset{A}{\exists}_{a_1,a_2} a_1 < a_2 \ \wedge\ \mathsf{A.pc}[a_1]{=}0$ and formula $\overset{A}{\exists}_{a_1} \mathsf{A.pc}[a_1]{=}\mathsf{A.pc}[x]$ are not pid quantified constraints. For the *unique existential quantifier* $\overset{A}{\exists}_{\boldsymbol{a}}$, we require that variables in set $\boldsymbol{a}$ are only used to index local variables of Module A, and each index variable should be mapped to a unique integer no greater than $\mathsf{A.Cnt}$, i.e. a state $s \models \overset{A}{\exists}_{a_1,\ldots a_n} expr$ iff. there exist a valuation v such that the state s satisfies $expr[a_1/v(a_1), ..., a_n/v(a_n)]$ and $\forall_{a_i \neq a_j} v(a_i) \neq v(a_j) \ \wedge\ v(a_i) \leqslant \mathsf{A.Cnt}$. For example, state $(2, 2, [1, 1, \perp, \ldots])$ satisfies formula $\overset{A}{\exists}_{a_1,a_2} \mathsf{A.pc}[a_1]{=}1 \wedge \mathsf{A.pc}[a_2] = 1$, but it does not satisfies $\overset{A}{\exists}_{a_1,a_2,a_3} \mathsf{A.pc}[a_1]{=}1 \ \wedge\ \mathsf{A.pc}[a_2]{=}1 \ \wedge\ \mathsf{A.pc}[a_3]{=}1$.

In verification a pid quantified constraint $\overset{A}{\exists}_{a} \ldots \overset{L}{\exists}_{l}\ expr$ is usually used to represent the set of states satisfying this formula. For example, $\overset{A}{\exists}_{i,j}\mathsf{A.pc}[i] \leqslant \mathsf{A.pc}[j]$ represents the system states such that there exist two processes of type A, where the local variable pc of one process is no greater than the other. In fact $\overset{A}{\exists}_{i,j}\mathsf{A.pc}[i]\leqslant\mathsf{A.pc}[j] \equiv \overset{A}{\exists}_{i,j} true$, and it might be amazing that $\overset{A}{\exists}_{i,j}\mathsf{A.pc}[i]+\mathsf{A.pc}[j]\neq 5 \equiv \overset{A}{\exists}_{i,j} true$. As far as we know, it is not clear the subset test of two pid quantified constraints is decidable. However in practice, we have efficient conservative algorithms (sufficient condition) to handle subset test.

Satisfiability of a pid quantified constraint is decidable, as the problem can be transformed into the satisfiability of Presburger formulas. A pid quantified constraint $\overset{A}{\exists}_{a} \ldots \overset{L}{\exists}_{l}\ expr$ is satisfiable if and only if $expr[\mathsf{A.v}[a]/A_{v_a}]$ is satisfiable. Here we replace every appearance of array elements with a free integer variable in $expr$, as shown by $\mathsf{A.v}[a]/A_{v_a}$. For example, $\overset{A}{\exists}_{i,j}(\mathsf{A.pc}[i]<\mathsf{A.pc}[j] \ \wedge \ \mathsf{A.pc}[j]<\mathsf{A.pc}[i])$ 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 instances. For example, 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, we can make the size of transition relation a 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.

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

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

The semantics of $t1$ is "if there exist a process of module 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 module A by 1 and initialize the new local variables.

Verification As discussed before PRE operator needs to be defined to model check CTL properties. Without loss of generality, we assume that transition actions are classified to two types, pure assignment action and process instantiation action. We now discuss how to compute PRE for these two types of actions.

Type I. For $T = (\mathbf{G}, \mathbf{P}, \mathbf{L})$, a Type I transition rule τ_A of Module A is expressed as

$$\tau_A := \overset{A}{\exists}_{i}\Big(\mathcal{T}(I^G, O^G, I^L_i, O^L_i) \ \wedge \ \textsc{same}(\hat{O})\Big) \quad (3)$$

$$\textbf{where } \textsc{same}(V) := \bigwedge_{v\in V} v' = v$$

In Equation 3 index variable i identifies the process to take action, set I^G, O^G, I_i^L, O_i^L represents the global input, output, local input and output variables respectively, obviously I_i^L and O_i^L contains local variables indexed by i only. $\hat{O} = \mathbf{L} \cup \mathbf{G} \cup \mathbf{P} - O^G - O_i^L$ 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=$\{a\}$, $I_i^L = O_i^L = \{\mathsf{A.pc}[i]\}$, and $\hat{O} = \{\mathsf{A.pc}[j] \mid j \neq i\} \cup \{\mathsf{A.Cnt}\}$.

We suppose that a set of system states S is represented by a pid quantified constraint $\overset{A}{\exists}_{\boldsymbol{a}} \ldots \overset{L}{\exists}_{\boldsymbol{l}}\ expr(L_s^A, L_s^B, ..., L_s^L, G_s)$, where G_s, L_s^A, ..., L_s^L are the sets of global variables and local variables of module A to L appeared in $expr$ respectively. Then PRE(S,τ_A) can be computed in two steps. First we compute the conjunction $\mathcal{C} = \mathcal{S}' \wedge \tau_{\mathcal{A}}$. Here the *next form* S' is to simply substitute all variables appeared in $expr$ with their "primed" forms. For example, the next form of $\overset{A}{\exists}_{a_i} \mathsf{A.pc}[a_i]$=1 is $\overset{A}{\exists}_{a_i} \mathsf{A.pc}[a_i]'$=1. 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 $\mathcal{C}$, $\text{PRE}(S, \tau_A) := \exists_{\mathcal{X}}\ \mathcal{C}$. Now the formula to compute the first step $\mathcal{C}$ is listed as follows.

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

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 $\text{PRE}(S, \tau_a)$.

Example 1. Let the current states S represented by formula a=1 $\wedge$ $\overset{A}{\exists}_{a_1} \mathsf{A.pc}[a_1]$=0, and the transition τ_A be $t1$ in Equation 1. Then S' is a'=1 $\wedge$ $\overset{A}{\exists}_{a_1} \mathsf{A.pc}[a_1]'$=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 $\mathsf{A.pc}[a_1]'$ 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.

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

Thus, after existential quantification, we get

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

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

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

As shown in the calculation steps above, we have two disjuncted clauses in $\mathcal{C}$. The first one handles the case when a_1 and $\mathsf{A.Cnt}'$ are mapped to two different numbers. The second clause handles the case when index variable a_1 is instantiated as $\mathsf{A.Cnt}'$ (we replace all appearance of a_1 in S' and take off the quantifier on a_1). It is obvious that the first clause is a subset of the second clause, and we 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 $\mathbf{EF}(a = 2)$ for the example listed in Fig. 1. As the set of states $\mathsf{EF}\ p$ is evaluated by fixpoint $\mu x.p \vee \text{PRE}(x, \tau)$. 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	$a{=}\ 2 \vee a{=}1 \wedge \overset{A}{\exists}_i \mathsf{A.pc}[i]{=}0$
EF2	$a{=}2 \vee a{=}1 \vee a{=}0 \wedge \overset{A}{\exists}_{i,j} \mathsf{A.pc}[i]{=}0 \wedge \mathsf{A.pc}[j]{=}0$
EF3	$a{=}2 \vee a{=}1 \vee a{=}0 \wedge \overset{A}{\exists}_i \mathsf{A.pc}[i]{=}0 \vee a{=}{-}1 \wedge \overset{A}{\exists}_{i,j,k} \mathsf{A.pc}[i]{=}0 \wedge \mathsf{A.pc}[j]{=}0 \wedge \mathsf{A.pc}[k]{=}0$
EF4	$a{=}2 \vee a{=}1 \vee a{=}0 \vee a{=}{-}1 \wedge \overset{A}{\exists}_{i,j} \mathsf{A.pc}[i]{=}0 \wedge \mathsf{A.pc}[j]{=}0 \vee$ $a{=}{-}2 \wedge \overset{A}{\exists}_{i,j,k,l} \mathsf{A.pc}[i]{=}0 \wedge \mathsf{A.pc}[j]{=}0 \wedge \mathsf{A.pc}[k]{=}0 \wedge \mathsf{A.pc}[l]{=}0$

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 computation of set EF in Fig. 5 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 $\mathbf{AG}(\forall_{p_1,p_2} f(p_1, p_2))$ can be verified, because set of states $\mathsf{EF}(\exists_{p_1,p_2} \overline{f(p_1, p_2)})$ 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, verification becomes complicated. For example, can the progress property $\mathbf{AG}(\forall_{pid} \mathsf{pc}[pid] = wait \Rightarrow \mathbf{AF}(\mathsf{pc}[pid] = Enter))$ 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. 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.

2. 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.
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. 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.
5. F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–163, 2001.
6. 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.
7. 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.
8. H. Davulcu, M. Kifer, C.R.Ramakrishnan, and I.V.Ramakrishnan. Logic based modeling and analysis. In *pods*, 1998.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. M. Gillmann, J. Weissenfels, G. Weikum, and A. Kraiss. Performance and availability assessment for the configuration of distributed workflow management systems. In *Proceedings of the 7th International Conference on Extending Database Technology*, LNCS 1777, pages 183–202, 2000.
15. 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.
16. 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.

17. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
18. 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.
19. 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.
20. A. Kraiss, F. Schön, G. Weikum, and U. Deppisch. Towards response time guarantees for e-service middleware. *IEEE Data Engineering Bulletin*, 24(1):58–63, 2001.
21. K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.
22. 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.
23. 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.
24. A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science*, 1977.
25. A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
26. R.E. Bryant. Graph-based algorithms for boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1986.
27. H. Saidi. Model checking guided abstraction and analysis. In *Proceedings of Statica Analysis Symposium*, Lecture Notes in Computer Science. Springer, 2000.
28. 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.
29. 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.
30. 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.
31. 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.