Reducing Component Systems' Behavior Specification

Viliam Holub
Distributed Systems Research Group
Charles University, Czech Republic
Email: holub@dsrg.mff.cuni.cz

Frantisek Plasil
Institute of Computer Science
Academy of Sciences of the Czech Republic
Email: plasil@cs.cas.cz, plasil@dsrg.mff.cuni.cz

Abstract—Behavior verification of large component systems suffers of state explosion in particular when components involve parallel activities. For behavior protocols, a method of component behavior specification, we present a method of state space size reduction based on symbolic manipulation with the specification done by applying a set of reduction rules. A case study is presented showing that the specification size is often reduced to only a fraction of the original one.

I. Introduction

A. Behavior protocols

Behavior protocols [1] were designed as a specific process algebra, to specify the desired finite sequences of method calls on component interfaces (their interplay) - the behavior of components.

A behavior protocol P is an expression that generates a set of traces of event tokens representing atomic events (actions) related to method invocations ($?m\uparrow$ stands for accepting a method m invocation, $!m\uparrow$ issuing an invocation of m, $?m\downarrow$ accepting the response (end) of m's execution, $!m\downarrow$ issuing the response). In addition to event token, P is composed of operators (';' sequencing, '+' alternative, '$*$' repetition, and '|' parallel interleaving without a communication), and abbreviations $?m$ (stands for $?m\uparrow$; $!m\downarrow$), $?m\{P\}$ (meaning $?m\uparrow$; P; $!m\downarrow$); similar rules are introduced for $!m$, and $!m\{P\}$.

The behavior protocol specifying the behavior of a particular component is called a *frame protocol*. As an example, consider the frame protocol of the AccountDatabase component depicted in Fig. 15 from Fig. 1. The protocol specifies that on its provided interface IAccount, it accepts a call of GenerateRandomAccountId, or alternatively ($+$), calls of CreateAccount and RechargeAccount. In the latter case as a reaction on accepting the call it issues a call of Withdraw on its required interface ICardCenter. This can be repeated a finite number of times ($*$). In parallel (|) to this, AccountDatabase can repeatedly accept calls on IAccount of AdjustAccountPrepaidTime_1, AdjustAccountPrepaidTime_2, and AdjustAccountPrepaidTime_3. Each time these adjusting calls can be accepted in a sequence a finite number of times.

Behavior protocols introduce special case of parallel composition (the consent operator $\bigtriangledown$) [2] with communication and hiding as known from the process algebra ACP [3]. It produces interleaving of events, while merging the invoke "!" and accept "?" events with the same name into an internal event "τ" which (similarly to CCS [4]) in principle means combining communication and hiding as defined in ACP. Moreover, accept events are blocking, while invoke events have to be merged by a counterpart immediately; unlike other process algebras, the consent operator produces specific event tokens corresponding to *composition errors* which are

- *bad activity* occurring when the issued event cannot be accepted,
- *no activity* (deadlock) when only accept events are enabled, and, since only finite traces are allowed,
- *divergence* (infinite activity) when the composition would produce an infinite trace.

By convention, a communication error is expressed by an error event token $!\epsilon$ for bad activity, $\oslash\epsilon$ for no activity, and $\infty\epsilon$ for infinite activity, which is always the final token in an erroneous trace [2].

B. Checking compliance

Because of its ability to identify communication errors, the consent operator is advantageously used to verify component behavior compliance. By composing the frame protocols of the communicating components on the same level of nesting (e.g. the frame protocols of ValidityChecker, CustomToken, and Timer in Fig. 1), it is verified that these components will cooperate correctly — *horizontal compliance* is verified. Naturally, this is true provided their implementation obeys their frame protocols.

In a similar vein, it is important to verify whether the composed behavior of the components cooperating at a particular level of nesting complies with the behavior specified for the surrounding (parent) component (Token in the example above). This *vertical compliance* is again verified with the help of the consent operator via the following trick: Even though the parent component in principle just mediates the calls (both incoming and outcoming) for its subcomponents, it can be easily turned into an 'environment' component which, instead of mediating, really issues and accept these calls. Its frame protocol is easily composed as the inverted frame protocol of the parent component, with ! replaced by ? and vice versa.

Going back to the example, the composition FPValidityChecker $\bigtriangledown$ FPCustomToken $\bigtriangledown$ FPTimer thus verifies the horizontal compliance and FPToken^{-1} $\bigtriangledown$ (FPValidityChecker $\bigtriangledown$ FPCustomToken $\bigtriangledown$ FPTimer) the vertical compliance. Here
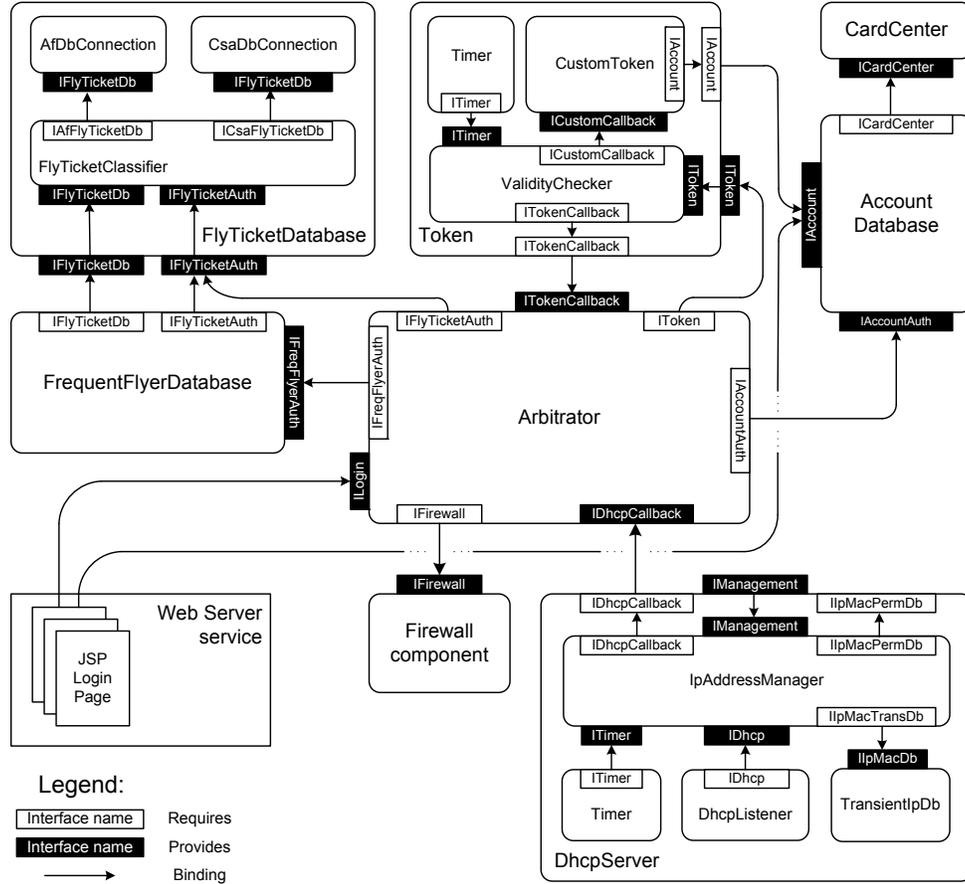
Fig. 1. Architecture of the Airport Internet providing service

FP stands for a frame protocol and $^{-1}$ denotes protocol inversion.

Verification of compliance is done by model checking — a number of specialized model checkers have been designed for this purpose [5], [6], [7]. Since parallel composition is involved via the operators | and $\bigtriangledown$, the Cartesian products of the state spaces associated with the operands of each of these operators tend to run into the *state explosion problem*.

C. Goals and basic idea of contribution

State explosion is a problem inherent to model checking involving parallel activities. The typical techniques to address it include abstraction [8], abstract interpretation [9], and partial order reduction [10], [11]. The former is hard to apply to behavior protocols' compliance verification, since the level of abstraction at with they capture behavior of software components is already very high (they abstract from component state and method parameters). At a first sight, partial order reduction is much more promising, since (i) the events in operands of the | operator do not communicate, however, their interleavings can significantly influence whether a non-blocking transition is enabled or not.

The goal of this paper is to show that state explosion in behavior compliance verification can be addressed by reducing the frame protocols via a technique which, in addition to (i),

employs observation that (ii) each pair of components communicates by events with unique, dedicated names (composed of the name of an interface and method). Advantageously, this can be employed to predict the communication leading to τ actions in the composition done by the consent operator. The reduction technique, reduction process (Sect. II), is based on a set of heuristic reduction rules (Sect. III). In some cases, the reduction can even eliminate the need of actual model checking, as the frame protocols involved in parallel composition get reduced to $NULL$. Since the original behavior specification (frame protocols) is modified, it is very important to find a way to interpret a counter example found by a model checked in the original frame protocols. This is shown in Sect. IV. The proposed reduction process was applied in a case study (Sect. V) with a positive experience (Sect. VI).

II. REDUCTION PROCESS

A. Overall strategy

The classical method of system verification via model checking consists of three steps. First, the user writes a specification (a model) of the system. Then, the specification is verified by a model checker. Finally, the result of the verification (report on correctness or a counter example) is
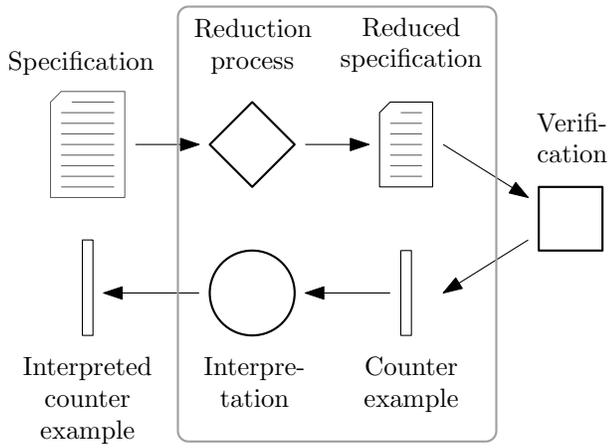
Fig. 2.   Overall strategy

Fig. 3.   The LTS representation of the parallel composition $!a\{?b\}\triangledown?a\{!b|!c;!c\}\triangledown?c*$

presented to the user. We add two more steps: reduction and counter example back interpretation (Fig. 2).

a) Reduction: Before running the model checker, the specification is analyzed and reduced (by a tool) in order to lower the size of the corresponding state space. In addition to the reduced specification, the tool produces also a log of the modifications done in the original specification. The reduced specification is then passed to the model checker.

A reduction requires a thorough knowledge of the relations among different concurrent activities involved in a consent operation. Therefore, with the aim to efficiently capture these relations, a frame protocol is represented as a hierarchy of LTSs, following the syntactical nesting of the | operators; roughly speaking, a frame protocol is represented as a hierarchy of parallel automata. Moreover, to capture potential communication in the consent operation, *counterpart* relation is maintained in addition. The basic idea is that pairs of the form $(!i.a\uparrow, ?i.a\uparrow)$ and $(!i.a\downarrow, ?i.a\downarrow)$ are in the relation. The conversion from textual frame protocols to this LTS internal representation is described in Sect. II-B.

The actual reduction is achieved by a repetitive application of reduction rules. Each reduction rule describes a frame protocol modification (its LTS modifications) and the set of conditions which must be satisfied in order to apply this rule. The list of reduction rules and the associated conditions are described in Sect. III.

b) Counter example back interpretation: As the model checker finishes by reporting an error in the reduced specification and not in the original one, the counter example has to be back interpreted for the user. This is achieved by applying inversion of all the modifications saved in the log; details are described in Sect. IV.

As the bottom line, the reduction is transparent to the user.

B. Converting Behavior protocol into LTS

As shown in Sect. I-A, the frame protocol of a component is an expression employing in addition to the classical regular expression operators +, ; and * also | and $\triangledown$. While it is easy to convert a regular expression into LTS, when converting a
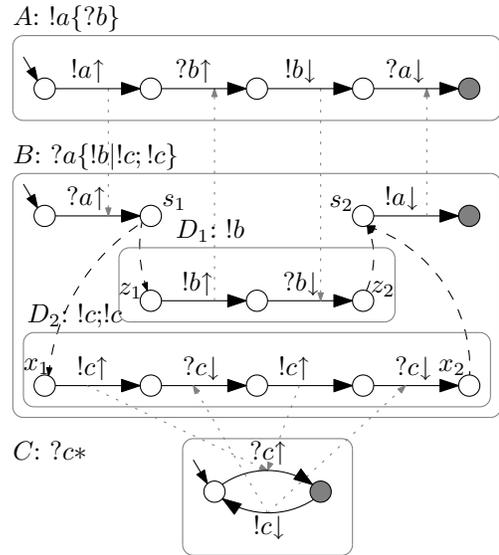
frame protocol a special care must be taken for the parallel operators | and $\triangledown$. The semantics of $A|B$ is defined as all the possible interleavings of the traces generated by the protocols A and B. Obviously, creating a corresponding LTS based on this definition typically leads to enormous size of the LTS — it determines a subset of Cartesian product of the state spaces of A and B. Therefore, because the events in A and B do not communicate, they can be actually expressed as separate, parallel LTSs (basically following the idea of parallel automata). In general, in a frame protocol fp these LTSs can be nested, being constructed recursively by following the syntactic structure of fp.

Consider an example of parallel composition of frame protocols of three components $!a\{?b\}\triangledown?a\{!b|!c;!c\}\triangledown?c*$, where the first component calls a method a and meanwhile accepts a callback b, the second component awaits a call of a and implements it as a parallel call of the methods b and c (two calls of the latter), and the third component accepts an arbitrary number of the c method calls. The corresponding LTSs working in parallel are depicted in Fig. 3.

We graphically capture LTSs as boxes with round corners (sometimes except for the topmost LTS). In the protocol $?a\{!b|!c;!c\}$ represented by the LTS B, the execution splits after $?a\uparrow$ from state s_1 in two nested LTSs D_1 and D_2, with the initial states z_1 and x_1. We say that states z_1 and x_1 are *associated* with s_1. By convention, associations are depicted by dashed arrows like those from the state s_1 to the states z_1 and x_1. In a similar vein, when both LTSs D_1 and D_2 finish in the states z_2 and x_2, the execution joins in the state s_2 and continues in the LTS B.

III. REDUCTION RULES

A. Elimination of ν-transitions

As mentioned in Sect. I-A, a τ event is created as a result of a parallel composition (via $\triangledown$) of an invoke and accept events with the same name; in some process algebras it is said that such two events *synchronize* or *communicate*. Assuming a τ event is produced in the result of $A\triangledown B$ by synchronizing $!a\uparrow$ and $?a\uparrow$, it becomes here an internal action which cannot synchronize any further. In particular, in a subsequent parallel composition such as $(A\triangledown B)\triangledown C$, this τ represents an "uninteresting" asynchronous action. However, its presence may be important to express that some other events are not immediately enabled; this holds for each of the parallel compositions $A\triangledown B$ and $(A\triangledown B)\triangledown C$. The key idea behind this paper is whether one could statically decide on eliminating the actions $!a\uparrow$ and $?a\downarrow$ directly from A resp. B (i.e. reduce A resp. B) for the purpose of compliance checking. Obviously, such an elimination should neither introduce a communication error, nor eradicate one. The challenge is to find the reduction rules which would guarantee this requirement. Since it is easier to articulate such rules for the LTS representation of a protocol, we will use this notation in the rest of this section.

Consider again the composition $A\triangledown B$ and event tokens $!a\uparrow$ resp. $?a\uparrow$ to appear in A resp. B. in such a way that they synchronize (which of the appearances can synchronize is determined from the counterpart relation).

To help articulate rules for making sure that elimination of such transition does not eradicate a communication error, consider first the following example:

$$(!a*|!b*)\triangledown(?a+?b)*$$

The corresponding LTS is in Fig. 4 where the abbreviations are expanded. By convention, the transition which are candidates for elimination are denoted as ν-transition — in this example these are the transitions originally labeled $?a\downarrow$ and $!a\downarrow$. Semantically, a ν-transition is an empty transition ($NULL$ in protocols) not visible in any trace. It is similar to the ϵ-transition in automata theory [12], where this transition accepts an empty input string. Notice, however, that if the ν-transition was eliminated in Fig. 4, the $\triangledown$ composition would eradicate the bad activity error present in the original composition $(!a*|!b*)\triangledown(?a+?b)*$. The error is caused by $!b\uparrow$ which cannot be accepted immediately when $!a\uparrow$ is accepted, since $!a\downarrow$ is to be issued first. The corresponding error trace is $\tau;!b\epsilon$. Obviously, the ν-transitions were not good candidates for elimination. Below are the rules for a save removal of a ν-transition.

Consider a ν-transition $\overrightarrow{s_1s_2}$. If it complies with the following rules, it is guaranteed that no communication error will be eradicated by $\overrightarrow{s_1s_2}$ removal:

(i) If s_2 has only outgoing transitions and $s_1 \neq s_2$ (Fig. 5a), the transition is removed and states s_1 and s_2 are joined into a single state $s_1 \circ s_2$.

(ii) If s_2 features both outgoing and incoming transitions other than the ν one (Fig. 5b), the ν-transition $\overrightarrow{s_1s_2}$ is
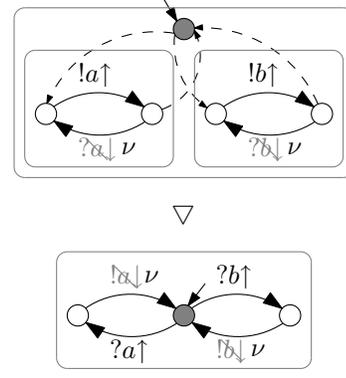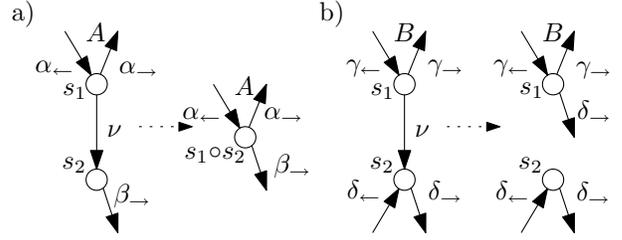

Fig. 4. Naive ν-transition elimination

Fig. 5. Elimination of ν-transition

removed and all the outgoing transitions from s_2 (such as δ) are duplicated by introducing them to s_1 as well.

(iii) If $s_1 = s_2$, the transition can be safely removed; this is a special case of (ii).

To justify the rules, assume a ν-transition $\overrightarrow{s_1s_2}$ was eliminated and a bad activity error $!a\epsilon$ was eradicated by that. Then the result of the elimination accepts an event a which would not be accepted in the original LTS. Therefore there has to be a transition outgoing from s_2 accepting a, but none such transition outgoing from s_1. Hence to avoid this bad activity error, all the accepted events in s_2 must be accepted in s_1 as well.

B. Identification of ν-transactions

Consider a parallel composition of the form $A_1\triangledown A_2$. This section provides a list of rules as to how to identify and eliminate ν-transitions from A_1 and A_2 in such a way that no communication error will be injected into this parallel composition. At the same time, applications of these rules assume that the conditions (i) - (iii) from Sect. III-A are satisfied. Again, the rules are articulated for the LTS representation of A_1 and A_2.

1) Simple method call: This rule (Fig. 6) addresses calls of a method a, assuming that no "reactions" via $\{\ldots\}$ for both accepting and issuing of such a call are specified in A_1 and A_2; i.e. each of the accepting call specification in A_2 takes the form $?a\uparrow;!a\downarrow$, and each issuing of such call in A_1 is specified as $!a\uparrow;?a\downarrow$.

Obviously, the basic idea is that, with respect to the consent operator, the transitions $?a\downarrow$ and $!a\downarrow$ are "redundant"
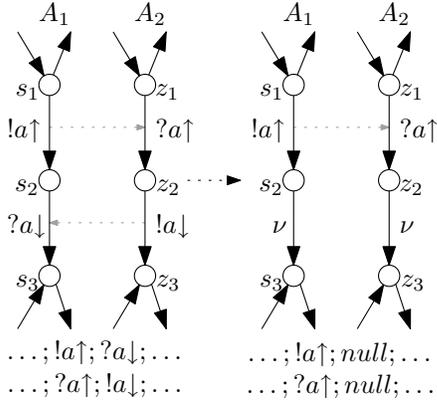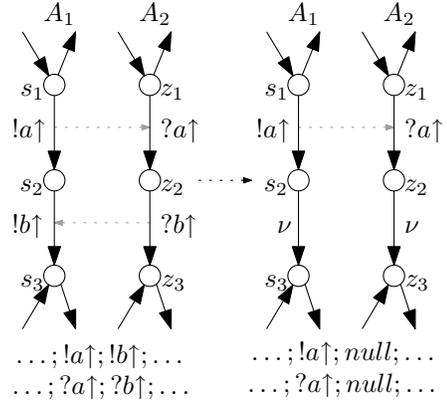
Fig. 6.   Simple method call

Fig. 7.   Serial method invocation

in the specification, can be replaced by ν-transitions, and reduced. The correspondence of the state triples (s_1, s_2, s_3) and (z_1, z_2, z_3) is determined by the counterpart relation. This reduction does not introduce a communication error provided:

(i) there are no other transition from the states s_2 and z_2
(ii) for each instance of the state triple (s_1, s_2, s_3), there is an instance of (z_1, z_2, z_3) determined by the counterpart relation (and vice-versa)
(iii) the reduction is performed for all instances of (s_1, s_2, s_3) and (z_1, z_2, z_3) satisfying (ii).

To show that no communication error is injected into A_1 and A_2 by the application of this rule, assume no communication error is present in $A_1 \triangledown A_2$ due to the parallel composition of (s_1, s_2, s_3) and (z_1, z_2, z_3). Since (i) requires no other transition from the states s_2 and z_2 to exist, skipping of $?a\downarrow$ and $!a\downarrow$ - events that communicate, cannot introduce a communication error.

2) Serial method invocation: A sequence of method invocations (Fig. 7) is often a result of simple method call reductions. The basic idea is that since $!a\uparrow$ and $?a\uparrow$ synchronize, the following events $!b\uparrow$ and $?b\uparrow$ are "redundant" and can be safely replaced by ν-transitions and removed. Again, the correspondence of the state triples (s_1, s_2, s_3) and (z_1, z_2, z_3) is determined by the counterpart relation, and this reduction does not introduce a communication error provided:

(i) there are no other transition from the states s_2 and z_2,
(ii) for each instance of the triple (s_1, s_2, s_3), there is an instance of (z_1, z_2, z_3) determined by the counterpart relation (and vice-versa)
(iii) the reduction is performed for all instances of (s_1, s_2, s_3) and (z_1, z_2, z_3) which satisfy (ii).

Also to show that no communication error is injected into A_1 and A_2 by application of this rule, the same arguments as in the III-B1 hold.

3) Simple cycle: Let the specification of A_2 contain just a single state z and a transition t, labeled $?a\uparrow$ which thus begins and ends in z (Fig. 8a). Then all the events $!a\uparrow$ in A_1

which are in counterpart relation with t synchronize (unless they are unreachable in A_1). Therefore they (and t) can be safely replaced by ν transitions and removed. Since these events communicate while A_2 remains in state z, removing these ν transitions cannot inject a communication error. A similar rule can be articulated for a $!b\downarrow$ transition (Fig. 8b).
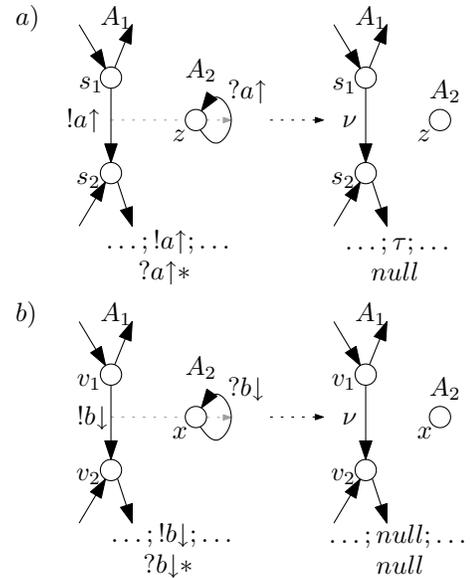

Fig. 8.   Simple cycle

4) External events: Consider verification of vertical compliance of two components with the frame protocols A and B. The components do not communicate only with each other, but also with their "environment" via their interfaces not yet bound (there are external events in the frame protocols). Note, that these external events are unambiguously identified, since their names are unique (Sect. I-C). Assume now that there is an ideal environment with the protocol E. Here "ideal" means that it (i) accepts any external event issued by A and B, (ii) issues any event A and B are ready to accept as external, and (iii) does not issue any other external event. Then, obviously, $E \triangledown (A \triangledown B)$ can contain

only such communication errors which are caused by the communication of A and B. Therefore, all the transitions labeled by an external event in A and B can be safely replaced by ν-transitions and removed (Fig. 9).
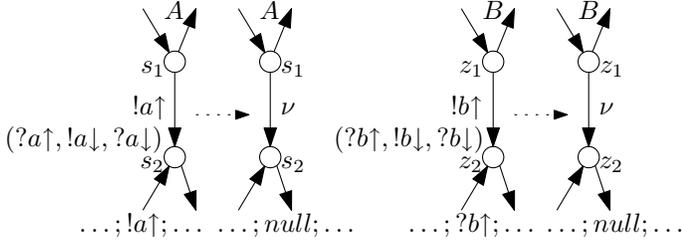

Fig. 9.   External events

5) Initital state transitions: Consider again a parallel composition $A_1 \triangledown A_2$. If from the initial states of A_1 and A_2 there are only single transitions which synchronize (Fig. 10), they can be safely replaced by ν-transitions and removed provided there is no "third-party" synchronization in the state z_2 (or s_2) as depicted in Fig. 11. In this setting, the ν-transitions from the initial state of A_1 and A_2 would eliminate the bad activity error ($!b\downarrow$ cannot be accepted immediately in the initial state of A_2.)
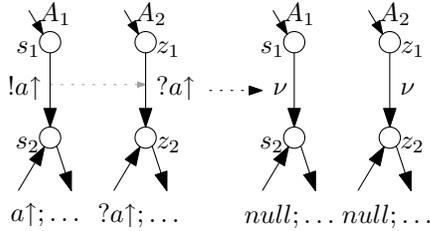

Fig. 10.   Initial state transition
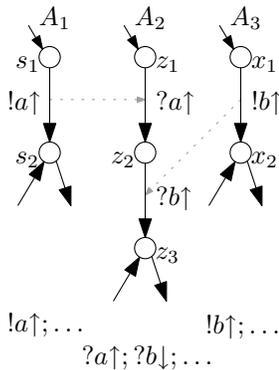

Fig. 11.   Danger of hiding a bad activity error

C. Reduction outside $\triangledown$ composition

This section articulates three rules for reducing an LTSs in a "classical" automata minimalization way.
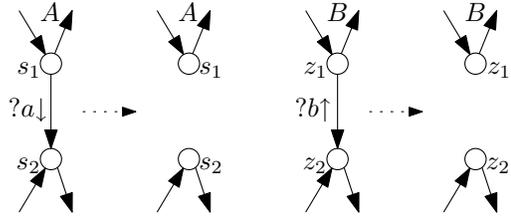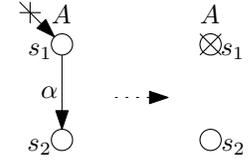

Fig. 12.   Unbound actions

Fig. 13.   Unreachable states

1) Unbound actions: Components often implement more business logic than required in a particular environment. Typically, the unused features are reflected in the component architecture as unbound interfaces. Obviously, in the frame protocol, no event at an unbound interface can be ever accepted, so that its acceptance never appears in a trace. Hence a transition labeled by such an accept event can be safely removed (Fig. 12). As an aside, on the contrary, issuing an event on unbound interface triggers a bad activity error in a consent composition.

2) Unreachable states: Obviously, unreachable states and consequently unreachable transitions are unnecessary in an LTS, since they do not contribute to any trace. At the same time, it is important to emphasize that reducing them may subsequently enable another reduction rules to be applied.

The basic idea of removing an unreachable states and all related transitions outgoing from it is illustrated in (Fig. 13). Here the state s_1 and also the transition α are removed in an LTS A. The unreachability of s_1 is emphasized by the crossed arrow. Unfortunately, to decide precisely whether a state is reachable (the reachability test) is time-consuming; typically it is as hard as the corresponding state space traversal itself. Therefore, for practical reasons, we impose a stronger condition in the reachability test: a state is unreachable if it is neither a starting state, nor an associated state, and has no incoming transition.

3) Redundant state: In general, by *redundant state* we mean a state which transitions (if any) do not contribute to any trace. However, for simplicity, we consider only four key situations when a redundant state is easily removable without introducing any communication error (Fig. 14). Even though the situations a) - d) in this figure look artificial, they typically result from a series of other reductions. Except for the trivial a) situation, the other reductions in Fig. 14 involve associate states related to nesting of LTSs in very special, pathologic situations, namely: A nested LTS contains just a single state and no transition (b), the outmost LTS contains just a single and associate state to initialize nested LTSs and another such

state for waiting the activities of these LTSs to be finished (c), and there is a redundant level of LTS nesting such as B in the case d).
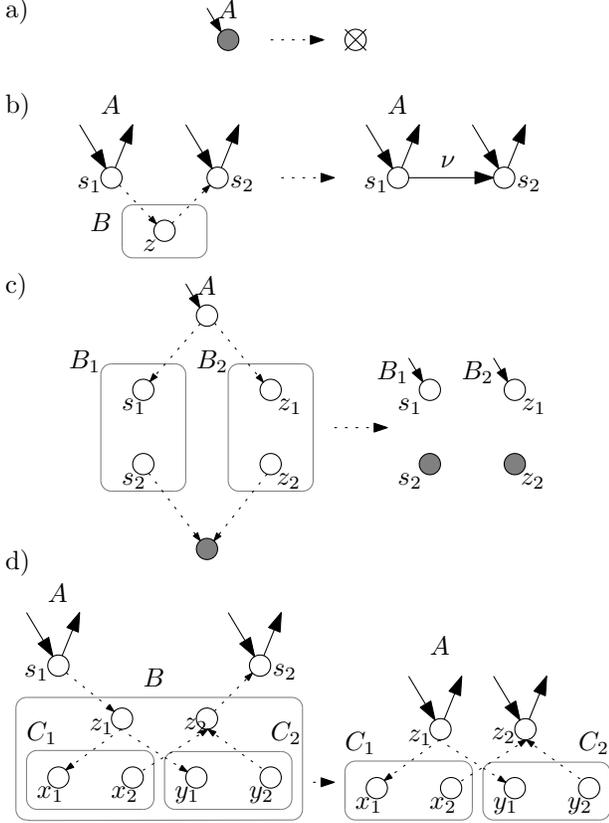

Fig. 14.   Redundant state

IV. COUNTER EXAMPLE BACK INTERPRETATION

For every application of a reduction rule, we save in a log file all the states and actions (transitions) that have been affected by it. To back interpret the counter example, we take all the reductions from the log in reverse order and apply them inversely to the states and transitions in the counter example to enhance it accordingly.

The reductions B1, B2, B3, and B4 replace unnecessary transition by a ν transition, which is immediately reduced according to Fig. 5 as discussed in Sect III-A. The counter example must be modified when a ν transition was (virtually) executed. Although the execution of a ν transition cannot be explicitly expressed in the counter example, its effect is indirectly recoverable from the transition's starting state. In Fig. 5a, it is the transition from $\beta_\rightarrow$ starting at $s_1 \circ s_2$. In Fig. 5b, it is the transition from $\gamma_\rightarrow$ starting at s_1.

The reduction B5 represents a direct action execution (τ). To back interpret its effect, we have to explicitly add the reduced action at the beginning of the counter example.

The reductions C1 and C2 remove unemployed behavior and thus are not reflected in the counter example. The reduction C3 modifies internal structures only, so that it is not reflected in the counter example either.

Step	Reduction	State space size	Time
0	Original specification	871122	143m42s
1	External events (26x)	36369	4m43s
2	Simple method call (10x)	9506	1m9s
3	Initial state transition(1x)	9504	1m4s
4	Redundant state (4x)	9504	1m4s
5	Simple cycle (9x)	108	0.4s
6	Simple method call (3x)	50	0.3s
7	Simple cycle (3x)	9	0.2s
8	Simple method call (2x)	4	0.2s
9	Simple cycle (2x)	1	0.2s
10	Redundant state (17x)	1	0.2s

V. CASE STUDY

In this section, we will share with the reader our experience with applying the reduction rules presented in Sect.III to a nontrivial demo application developed in one of our projects [13] (Fig. 1).

As a proof of the concept, we have taken one of the tests specifications prepared in the project (Fig. 15). The test consists of a consent composition of the following components: $Arbitrator, Token, AccountDatabase, CardCenter,$ and $Firewall$. Without applying the rules, the verification required to visit 871122 states and it took about 2 hours and 23 minutes in total to traverse them; see Tab. I. The table also illustrates the effect of applying the reduction rules.

First, external events (26 alltogeher) were to be replaced by ν. Ten of them could be removed immediatelly, such as $?IArbitratorLifetimeController.Start\uparrow$ and $?Ilogin.GetTokenIdFromIpAddress\uparrow$. However, removal of the rest of them was prevented by conditions of ν-elimination (Sect.III-A); in particular by the fact, that no accept transition can begin from the state which is the end of a ν-transition. Fortunately, these "preventing" transitions were also external events which could be removed. After all of these reductions, the verification took less than 5 minutes and required approximately 36000 states to traverse.

In the next step (2), ten simple method calls were reduced, for example $IFirewall.DisablePortBlock$ and $ICardCenter.Withdraw$. After that, in the step (3) initial state transitions $!ITokenLifetimeController.Start\uparrow$ and $?ITokenLifetimeController.Start\uparrow$ were eliminated. The status of the specification after this step finished is on Fig.16.

Further, the topmost redundant states in the components $Arbitrator$, $Token$, $AccountDatabase$ and $Firewall$ were reduced by applying the rule in (Fig. 14c). After this reduction, the number of the topmost LTSs rose from five to 17. In the next step (5), nine simple cycles such as $?IcardCenter.Withdraw\uparrow*$ were reduced. At the first sight surprisingly, after applying additional reduction rules (steps 6-10 in Tab. I) only a single state for each LTS remains, and is finally removed as redundant.

Arbitrator

```
( ?IArbitratorLifetimeController.Start^ ;
  !ITokenLifetimeController.Start^ ;
  [?ITokenLifetimeController.Start$,
    !IArbitratorLifetimeController.Start$] );
(
  (
    ?ILogin.GetTokenIdFromIpAddress +
    ?ILogin.LoginWithFlyTicketId {
      !IFlyTicketAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.LoginWithFrequentFlyerId {
      !IFreqFlyerAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.LoginWithAccountId {
      !IAccountAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.Logout {
      !IToken.InvalidateAndSave_1 }
  )* |
  ?ITokenCallback.TokenInvalidated_1 {
    !IFirewall.EnablePortBlock_1 }*
  |
  ?ITokenCallback.TokenInvalidated_2 {
    !IFirewall.EnablePortBlock_2 }*
  |
  ?ITokenCallback.TokenInvalidated_3 {
    !IFirewall.EnablePortBlock_3 }*
  |
  ?IDhcpCallback.IpAddressInvalidated_1 {
    !IToken.InvalidateAndSave_2 }*
)
```

Token

```
?ITokenLifetimeController.Start ;
( ?IToken.InvalidateAndSave_1 {
    (!IAccount.AjustAccountPrepaidTime_1 + NULL);
    !ITokenCallback.TokenInvalidated_1 }* |
  ?IToken.InvalidateAndSave_2 {
    (!IAccount.AjustAccountPrepaidTime_2 + NULL);
    !ITokenCallback.TokenInvalidated_2 }* |
  (
    (!IAccount.AjustAccountPrepaidTime_3 + NULL);
    !ITokenCallback.TokenInvalidated_3 )*
)
```

AccountDatabase

```
( (
    ?IAccount.GenerateRandomAccountId +
    ?IAccount.CreateAccount +
    ?IAccount.RechargeAccount {
      !ICardCenter.Withdraw }
  )* |
  ?IAccount.AjustAccountPrepaidTime_1* |
  ?IAccount.AjustAccountPrepaidTime_2* |
  ?IAccount.AjustAccountPrepaidTime_3*
)
```

CardCenter

```
( ?ICardCenter.Withdraw* )
```

Firewall

```
( ?IFirewall.EnablePortBlock_1* |
  ?IFirewall.EnablePortBlock_2* |
  ?IFirewall.EnablePortBlock_3* |
  ?IFirewall.DisablePortBlock*
)
```

Fig. 15.   System specification: arbitrator.bp

Arbitrator

```
(
  (
    (!IFirewall.DisablePortBlock^ + NULL)
    +
    (!IFirewall.DisablePortBlock^ + NULL)
    +
    (!IFirewall.DisablePortBlock^ + NULL)
    +
    !IToken.InvalidateAndSave_1
  )*
  |
  ?ITokenCallback.TokenInvalidated_1 {
    !IFirewall.EnablePortBlock_1^ }*
  |
  ?ITokenCallback.TokenInvalidated_2 {
    !IFirewall.EnablePortBlock_2^ }*
  |
  ?ITokenCallback.TokenInvalidated_3 {
    !IFirewall.EnablePortBlock_3^ }*
  |
  !IToken.InvalidateAndSave_2*
)
```

Token

```
(
  ?IToken.InvalidateAndSave_1 {
    (!IAccount.AjustAccountPrepaidTime_1^ + NULL);
    !ITokenCallback.TokenInvalidated_1 }*
  |
  ?IToken.InvalidateAndSave_2 {
    (!IAccount.AjustAccountPrepaidTime_2^ + NULL);
    !ITokenCallback.TokenInvalidated_2 }*
  |
  (
    (!IAccount.AjustAccountPrepaidTime_3^ + NULL);
    !ITokenCallback.TokenInvalidated_3 )*
)
```

AccountDatabase

```
( !ICardCenter.Withdraw^* |
  ?IAccount.AjustAccountPrepaidTime_1^* |
  ?IAccount.AjustAccountPrepaidTime_2^* |
  ?IAccount.AjustAccountPrepaidTime_3^*
)
```

CardCenter

```
( ?ICardCenter.Withdraw^* )
```

Firewall

```
( ?IFirewall.EnablePortBlock_1^* |
  ?IFirewall.EnablePortBlock_2^* |
  ?IFirewall.EnablePortBlock_3^* |
  ?IFirewall.DisablePortBlock^*
)
```

Fig. 16.   Situation after step 4

VI. Discussion and Related Work

Even though the results of the case study are persuasive, two obvious questions have to be answered to claim a real benefit of the presented reduction process.

a) In which order are the rules to be applied (and how many times): Based on experiments, the rule of thumb is that the Rules B4 and C1 reflect are to be applied first and only once (in any order). This is because they are driven by the static component architecture and do not consider actual relationship among method calls.

On the other hand, all the remaining rules depend on each other. For example, applying the rule B1 on two consecutive method calls will enable the rule B2 or B3 to be applied. At the same time, applying the rule B2 may enable B3 while applying the rule B3 may enable B1 and B4.

In general, it is not easy to specify a correct order of reductions to achieve minimal state space. However, for a class of specifications created by method abbreviations only, the best result is achieved by the same ordering of reduction rules as chosen in Sect. III. Fortunately, behavioral constructs outside the scope of method calls or acceptances are rare.

b) Is the result of reduction always a composition of empty protocols, similar to the case study: It is relatively easy to show that the answer is no. Consider for instance the composition of the protocols: $(?c\uparrow; !d\uparrow)*; ?a\uparrow \triangledown (?d\uparrow; !c\uparrow)*; !a\uparrow$ — there is no way to reduce them by static analysis. However, our experiments indicate that in all real case studies we had available, there is always a substantial reduction in the state space size, in particular when the B1 rule is repeatedly applied. On our future work list, there is the search for frame protocol classes which guarantee emptiness of the reduction result.

Related to our work are slicing and symmetry exploiting methods. In *program slicing*, we define *slicing criterion* which is typically a pair of program location and a set of variables. Then, a program slice is a part of the original program which affects the values of presented variables at the specified location.

The idea of program slicing was introduced by Weiser [14], [15] originally for debugging purposes. Later, program slicing has been used for various purposes such as analysis, parallelization, comparison, and testing. With respect to our work, the most interesting topic is compiler optimization. Larus and Chandra [16] present a detection of redundant common sub-expressions. The code is enriched by instructions for trace inspection. Traces are then interpreted as a stream of events, where a directed acyclic graph is constructed with all the arguments and operators which affects the current value in the register. Because the approach uses information about executions, it represents a class of *dynamic* slicing.

Specification slicing aims at creating a reduced specification while preserving all the desired information. It is analogous to *program slicing* for specifications. Wu and Yi [17] present slicing of Z [18] specifications. First, data, control, and logic dependencies of the specification are represented by a specification dependence graph. Then, the reduced Z specification is created by a two-pass reachability algorithm applied to the graph.

Another work (done in our group) [19] presents slicing of the component behavior specification according to the actual component composition. The technique aims at removing the unused behavior to make the actual real role of a particular component more visible by removing the unemployed parts of the frame protocols. Thus, although the size of the specification is reduced, the real size of the state space is untouched, since only its employed part contributes to the size of the consent composition.

A key difference between slicing and our reduction method is that slicing reduces the unemployed part of the specification, while our method is specialized towards consent composition and reduces also the parts of the specification for which composition without communication errors can be statically guaranteed. At the same time, the method guaranties that, by the reduction, no communication errors will be injected and no communication error will be eliminated from the specification. Moreover, our reduction method involves also logging of the partial reduction steps to ensure the original form of the specification can be reconstructed - this is necessary for providing counter examples referring to the original form if the reduction does not result in an empty specification and model checking has to be applied in its rest. This is not strictly required in a case of slicing.

There are several approaches to exploit symmetry in the specification. For example in Murϕ model checker [20], Ip a Dill introduced [21] a new data type *scalarset* which represents and unordered set. Operations on scalarset are restricted to guarantee that every function on the set is automorphism. Thus, scalarset is symmetric and the behavior of the program is independent on the actual permutation of elements. Additionally, conditions under which the scalarset can be used are statically checked.

VII. Conclusion

We attack the state space explosion problem by reducing the specification to be verified. The reduction is done by an iterative application of reduction rules, which have been articulated with respect to the consent composition. To guarantee that the reductions do not inject additional compositional errors and do not eradicate a compositional error present in the specification, we have formulated several conditions to be satisfied. Reductions are of a low overhead, since the algorithm is linear with the size of the specification.

Although a "full" reduction of the specification is not guaranteed in general, the real-life case studies, such as the one presented in this paper, show that most of such specifications contain several typical patterns to which reduction rules apply. If the result of the reductions is not an empty protocol and, therefore, a standard model checking verification is to be applied on the rest, the potential counter example is back interpreted in the original specification. This makes the reduction method fully transparent to the user.

REFERENCES

[1] F. Plasil, S. Visnovsky, and M. Besta, "Bounding component behavior via protocols," in *TOOLS*, vol. 30. USA: IEEE Computer Society, Aug 1999, pp. 387–398.

[2] J. Adamek and F. Plasil, "Component composition errors and update atomicity: Static analysis," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17(5), pp. 363–377, Sep 2005.

[3] J. A. Bergstra and J. W. Klop, "Process algebra for synchronous communication," *Information and Control*, vol. 60, no. 1-3, pp. 109–137, 1984.

[4] R. Milner, *A Calculus of Communicating Systems*, ser. Lecture Notes in Computer Science. Springer, 1980, vol. 92.

[5] M. Mach, F. Plasil, and J. Kofron, "Behavior protocol verification: Fighting state explosion," in *International Journal of Computer and Information Science*, vol. 6. ACIS, Mar 2005, pp. 22–30.

[6] P. Parizek, F. Plasil, and J. Kofron, "Model checking of software components: Combining Java PathFinder and behavior protocol model checker," Charles University, Tech. Rep. 2006/2, Jan 2006.

[7] V. Holub and P. Tuma, "Streaming state space: A method of distributed model verification," in *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*. IEEE Computer Society, 2007, pp. 356–368.

[8] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, 1977, pp. 238–252.

[9] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, Sep 1994.

[10] D. Peled, "Combining partial order reductions with on-the-fly model-checking," in *Proceedings of the 6th International Conference on Computer Aided Verification*, ser. Lecture Notes In Computer Science, D. L. Dill, Ed., vol. 818. Springer-Verlag, Jun 1994, pp. 377–390.

[11] A. Valmari, "A stubborn attack on state explosion," in *Proceedings of the 2nd Workshop on Computer Aided Verification (CAV'90)*, ser. Lecture Notes in Computer Science, E. M. Clarke and R. P. Kurshan, Eds., vol. 531. London, UK: Springer-Verlag, Jun 1991, pp. 156–165.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

[13] P. Jezek, J. Kofron, and F. Plasil, "Model checking of component behavior specification: A real life experience," in *International Workshop on Formal Aspects of Component Software (FACS'05)*, vol. 160. Elsevier B.V, Aug 2006, pp. 197–210.

[14] M. Weiser, "Program slices: formal, psychological, and pratical investigations of an automatic program abstraction method," Ph.D. dissertation, University of Michigan, 1979, ann Arbor.

[15] ——, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[16] J. R. Larus and S. Chandra, "Using tracing and dynamic slicing to tune compilers," University of Wisconsin-Madison, Tech. Rep. CS-TR-1993-1174, 1993.

[17] F. Wu and T. Yi, "Slicing Z specifications," *SIGPLAN Notices*, vol. 39, no. 8, pp. 39–48, 2004.

[18] M. Spivey, *Z Notation*. Prentice Hall, Jun 1992.

[19] O. Sery and F. Plasil, "Slicing of components' behavior specification with respect to their composition," in *10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2007)*, Jul 2007.

[20] D. L. Dill, "The Murφ verification system," in *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1996, pp. 390–393.

[21] C. N. Ip and D. L. Dill, "Better verification through symmetry," in *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1993, pp. 97–111.