Slicing of Components' Behavior Specification with Respect to their Composition[*]

Ondřej Šerý[1], František Plášil[1,2]

[1]Charles University in Prague, Faculty of Mathematics and Physics
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{sery, plasil}@dsrg.mff.cuni.cz
http://dsrg.mff.cuni.cz

[2]Academy of Sciences of the Czech Republic
Institute of Computer Science
plasil@cs.cas.cz
http://www.cs.cas.cz

Abstract. Being an important means of reducing development costs, behavior specification of software components facilitates reuse of a component and even reuse of a component's architecture (assembly). However, since typically only a part of the components' functionality is actually used in the new context, a significant part of the behavior specification may be superfluous. As a result, it may be hard to see (and filter out) the actual interplay among the components in their behavior specification. This paper targets the problem in the scope of behavior protocols [15]. It presents a technique for slicing behavior protocols with respect to a given context (composition), designed to remove the unused behavior from a behavior specification. The technique is based on a formal foundation, generic enough to support slicing with respect to a property expressed as a predicate. To demonstrate viability of the proposed approach, a positive experience with behavior specification slicing applied in real-life case study is shared with the reader (along with a short description of a prototype).

Keywords: **C**omponents-based software engineering, Behavior specification, Software architecture reuse.

1 Introduction

When reusing a software component (such as a COTS component) in a component-based application, it is likely that only a part of its functionality will be actually used. Assuming a behavior specification of the component is available, a significant part of it may be superfluous in the given application. As a result, it is hard to read and comprehend the actual interplay among the components from their behavior

[*] This work was partially supported by the Grant Agency of the Czech Republic project 201/06/0770.

specification. A similar issue arises, when reusing a component architecture (assembly) in a new environment, employing only a part of the provided functionality. In order to enhance readability of the behavior specification and facilitate human comprehension of the actual interplay among the components, it is desirable to find methods for slicing the behavior specification to make it contain only the parts really used in a specific component composition and/or environment.

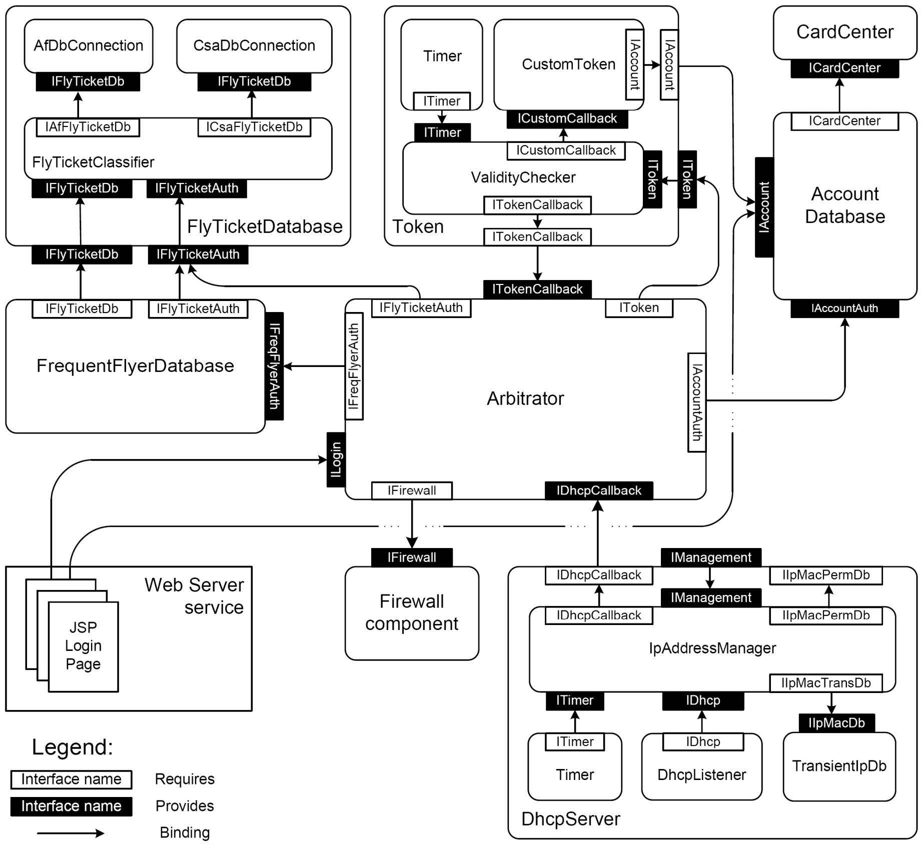

Fig. 1. Architecture of the demo component application, the airport internet providing service

The problem can be illustrated on a demo component application (Fig. 1), designed as a part of the CRE project [5]. The application constitutes an airport service for providing a wireless internet connection (to the owners of first or business class tickets, to the frequent flayer and credit card holders). As to the top-level components, `Firewall` realizes the firewall for blocking unauthorized internet connections by redirecting them to the login web page. The `FlyTicketDatabase` and `FrequentFlyerDatabase` components mediate access to the databases of airlines companies. `CardCenter` communicates with the bank credit card services and `AccountDatabase` encapsulates accounts with prepaid internet connection. The `Token` component is a dynamically created entity representing a single logged user. All communication is orchestrated by `Arbitrator`, while `DhcpServer` manages dynamic IP address allocation with support for use of the permanent IP address database. This database, mapping Mac to IP addresses, could be connected via the `IIpMacPermanentDb` interface and its use be triggered on via the `IManagement` interface. However, both these interfaces are left unbound, since permanent IP address allocation is not used in the application. The behavior of about twenty components was specified via the formalism of behavior protocols [15] and, using a behavior protocol checker, the behavior compliance of these components was verified.

Assume now an internet provision service in a public garden (payment by credit card only). Evidently, the component architecture in Fig. 1 can be reused for such a purpose. A necessary modification involves simplification of `DhcpServer` (no permanent addresses), and of `Artibrator` (no airport-specific logins). Obviously, it would be very confusing to see any `FlyTicketDatabase` and `FrequentFlyerDatabase`-related behavior in the specification of these modified components. Thus, slicing of behavior specification with respect to actual component architecture (composition) is desirable.

1.1 Behavior protocols

The formalism of behavior protocols [15] was developed for behavior specification of software components. As a behavior, the desired finite sequences of method calls on component's interfaces (their interplay) are considered, abstracting from method parameters and internal data. Behavior protocol specifying behavior of a particular component is called its *frame protocol*.

Being a specific process algebra [3], behavior protocol P is an expression that generates a set of traces of method calls (the language $\mathscr{L}(P)$). More precisely, a trace is a sequence of tokens representing atomic events related to method invocations (?a↑ stands for accepting a method invocation, !a↑ issuing an invocation, ?a↓ means accepting the response (end) of a method execution, !a↓ means issuing the response). Syntactically, a behavior protocol is composed of tokens, operators (";" sequencing, "+" alternative, "*" repetition, and "|" parallel interleaving), and abbreviations ?a (stands for ?a↑; !a↓), ?a{P} (stands for ?a↑; P; !a↓), and similarly for !a, and !a{P}.

In Fig. 2, there is a frame protocol of the `DhcpServer` component from Fig. 1. On its required interface `IDhcpCallback`, it calls `IpAddressInvalidated` to inform that an IP address lease has expired. This call can be repeated in parallel ("|") with accepting calls on the `IManagement` provided interface to mode switch between random IP address assignment and persistent MAC to IP address mapping (`!IIpMacPermDb.GetIpAddress*`). Since request for stopping permanent address assignement can come while `!IIpMacPermDb.GetIpAddress*` is in progress ("|"), this has to be captured by explicitly stating requests and responses of the mode switching calls.

Behavior protocols introduce special case of parallel composition (known from process algebras), the *consent operator* ∇. Similar to parallel composition e.g. in CCS, the consent operator produces interleaving of events, while merging the invoke "!" and accept "?" events with the same name into an internal event (prefixed by "τ"). Moreover, the consent operator identifies communication errors: *bad activity* – the issued event cannot be accepted, *no activity* (deadlock) – all of the ready events' tokens are prefixed by "?", and *infinite activity* (divergence) – the composed protocols "cannot reach their final events at the same time", so that the composed behavior would contain an infinite trace (only finite traces are allowed). Technically, these communication errors are reflected by ∇, appending the erroneous traces with error events (!ε, Øε, and ∞ε for bad activity, no activity, and infinite activity errors, respectively). For more information, the reader is referred to [1].

```
(
  !IDhcpCallback.IpAddressInvalidated*
  |
  (
    ?IManagement.UsePermanentIpDb↑ ;
    (
      !IIpMacPermDb.GetIpAddress*
      |
      (
        !IManagement.UsePermanentIpDb↓ ;
        ?IManagement.StopUsingPermanentIpDb↑
      )
    ) ;
    !IManagement.StopUsingPermanentIpDb↓
  )*
)
```

Fig. 2. DhcpServer frame protocol

To illustrate use of the consent operator, suppose that the architecture of the composed `DhcpServer` component is to be checked for correctness of its internal communication—*horizontal compliance*. This is achieved by applying the consent operator to the frame protocols of all subcomponents of `DhcpServer` and finding out whether the resulting language: $\mathscr{L}(FP_{\text{IpAddressManager}} \nabla FP_{\text{DhcpListener}} \nabla FP_{\text{TransientIpDb}} \nabla FP_{\text{Timer}})$, contains any erroneous trace. Another important task is to check whether the architecture of the composed `DhcpServer` component obeys its frame protocol—*vertical compliance*. For this purpose the *inverted frame* trick is used. The frame protocol of `DhcpServer` is inverted ("?" are substituted by "!" and vice versa) and the inverted protocol is composed with the protocol of the architecture: $\mathscr{L}(FP_{\text{DhcpServer}}^{-1} \nabla FP_{\text{IpAddressManager}} \nabla FP_{\text{DhcpListener}} \nabla FP_{\text{TransientIpDb}} \nabla FP_{\text{Timer}})$. The result is then sought for erroneous traces. The key idea behind the trick is testing the architecture in the most general environment of `DhcpServer`, represented by its inverted frame protocol $FP_{\text{DhcpServer}}^{-1}$.

1.2 Goal and structure of the paper

The goal of the paper is to propose a way of reducing frame protocols of components with respect to a particular component composition (architecture/assembly) in order to omit the unused parts of the behavioral specification. This should clarify the actual role of each component in their composition and make understanding of the overall behavior interplay of the components easier.

This goal is reflected in the structure of the paper as follows. Formal foundation of behavior protocol reductions and protocol slicing is provided in Sect. 2, while Sect. 3 introduces slicing with respect to composition and proposes a technique to achieve this kind of protocol reduction. The last sections are devoted to a prototype's description, related work discussion, and a conclusion.

2 Reduction and slicing of behavior protocols

2.1 Reduction preorder

First of all, it is necessary to formalize the notion of reduction; i.e. to define when a behavior protocol can be considered a reduction of another one. For this purpose, the notion of *substitutability* of components (and their behavior protocols) is crucial. Suppose that a component B working in an environment Env_B without any communication errors (Fig. 3-a) is to be substituted by another component A and each of them is associated with its frame protocol.

Definition 1. A behavior protocol a is *substitutable* for a behavior protocol b, if $\mathscr{L}(a \nabla b^{-1})$ does not contain any trace with communication error. A component A is *substitutable* for a component B, if the frame protocol of A is substitutable for the frame protocol of B.

In other words, Def. 1 says that a component A is substitutable for another component B, if by placing A to the most general environment of B (described by the inverted protocol b^{-1}) does not result in any communication error. Thus A can be safely placed into any environment Env_B of B (Fig. 3-b), assuming B is working without any communication errors in this environment.

Having the substitutability defined, the next step is to formalize the reduction itself. The basic idea is as follows: A component B_{red} with a reduced frame protocol b_{red} working in an environment Env_{red}, can be replaced by a component B with the frame protocol b, provided b is substitutable for b_{red} and $\mathscr{L}(b_{red}) \subseteq \mathscr{L}(b)$, Fig. 3-c. This is captured by defining a *reduction preorder* $\leq_R$ over behavior protocols in Def. 2. For the proof that the relation $\leq_R$ is really a preorder, the reader is referred to [17].

Definition 2. Let b_{red} and b be behavior protocols. b_{red} is *reduction* of b, $b_{red} \leq_R b$, if b is substitutable for b_{red} and $\mathscr{L}(b_{red}) \subseteq \mathscr{L}(b)$.

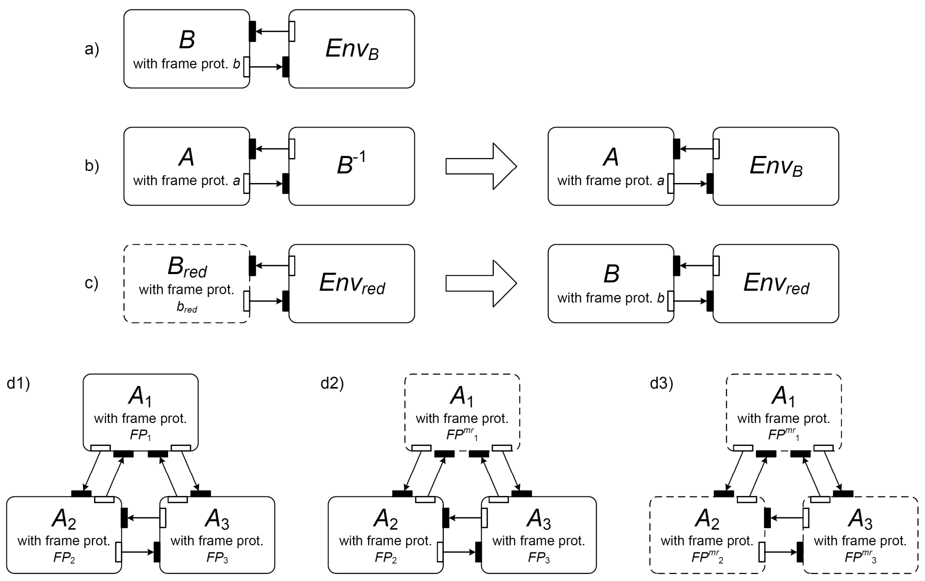

Fig. 3. Motivation for protocol reduction basic definitions

Intuitively, the reduction b_{red} of b is obliged to describe only a subset of the traces described by the protocol b ($\mathscr{L}(b_{red}) \subseteq \mathscr{L}(b)$). Furthermore, the original protocol b has to be substitutable for its reduction b_{red}. This is required to assure that any architecture (Env_{red}) designed using the reduced virtual component B_{red} can safely use any component B described by the original protocol b instead of B_{red}. Thus the designer can safely use the simpler reduced protocol b_{red} and be sure that the resulting architecture will function correctly even with a component featuring the more complex original protocol b.

2.2 Minimal reduction

The reduction preorder presented in Sect. 2.1 constitutes a formal instrument for deciding whether a protocol is a valid reduction of another one and for formalizing a set of valid reductions of a given protocol. However, the typical requirement is to find "in some sense" the minimal reduction of a given protocol. Furthermore, there is usually a constraint the resulting reduction should satisfy, so the goal is to find the minimal reduction satisfying the constraint (represented as a predicate C over languages in Def. 3). What the constraint actually is follows from the concrete type of the reduction, e.g. there is a set of important traces that should be preserved in the reduction (the actual constraint predicate for reduction with respect to composition is to be discussed later in Sect. 3). The minimal reduction could be straightforwardly defined as follows.

Definition 3. Let b be a behavior protocol over the alphabet Σ and C a predicate over 2^{Σ^*} (the constraint). A behavior protocol b_{red} is a *minimal reduction* of b satisfying C, if $b_{red} \leq_R b$, $C(\mathscr{L}(b_{red}))$ holds, and there is no behavior protocol c such that $c \leq_R b_{red}$, $C(\mathscr{L}(c))$ holds, and $\mathscr{L}(c) \subset \mathscr{L}(b_{red})$.

Even though such a definition seems natural, there are several issues to address: First, the minimal reduction is generally not unique, i.e. more minimal reductions can exist (even infinitely many, since, e.g. $\mathscr{L}$(a) = $\mathscr{L}$(a+a+…)). Second, the actual syntactical form of the protocols is not considered—the semantics of reduction is based on the languages generated by the protocols only. Third, if one tries to address the first two issues by requiring the minimal reduction to be the shortest one, then finding such a minimal reduction is a PSPACE complete problem. This follows from the close relation between regular expressions and behavior protocols and the well known fact that minimizing a regular expression is a PSPACE complete problem [7], [10], and [12] (a full-fledged justification is out of scope of the paper).

These observations trigger the need to develop a technique for finding reductions that would: i) assure uniqueness of the result, ii) take the actual syntactical form of the protocols into consideration, iii) be of a "reasonable" computational complexity, and iv) adhere to the reduction preorder as defined in Sect. 2.1. Such a technique – *protocol slicing* – is proposed in Sect. 3. It is based on the *slice* concept:

Definition 4. Let a and b be behavior protocols. We say a is *slice* of b, if a is reduction of b ($a \leq_R b$) and the syntax tree of a is derived by pruning the syntax tree of b.

In other words, the protocol reduction concept is based on the languages generated by the protocols, whereas protocol slicing brings into account also the syntactical form of the protocols by pruning the syntactical tree of the protocol to be reduced. In consequence, given a protocol b and a constraint C, there can be no its minimal reduction being also a slice of b. For instance, consider the protocol ?a* and the constraint that the method a will be called sequentially three times (more formally, the predicate C is defined as $C(\mathscr{L}) \equiv (\text{<?a; ?a; ?a>} \in \mathscr{L})$, where $\mathscr{L}$ is a language), then the protocol ?a* can be minimally reduced to ?a; ?a; ?a. On the other hand, there are only two slices of the protocol: NULL and ?a*, i.e. it is either sliced to empty protocol (which does not satisfy the constraint of three ?a), or remains unmodified. However, in general, slicing is practically more important than "optimal, language-based" reduction, since the former inherently means simplification of a protocol, while the latter can result in a blow-up of the protocol. For example, consider again the `DHCPServer` frame protocol in Fig. 2 and assume that the repetition on the lines 4–15 is to be repeated 3 times. The corresponding minimal reduction takes the form depicted in Fig. 4. Obviously, this reduction result becomes hard to read.

Definition 5. Let a be a behavior protocol over the alphabet Σ and C a predicate over 2^{Σ^*} (the constraint). A behavior protocol b is a *minimal slice* of a satisfying C, if b is a slice of a, $C(\mathscr{L}(b))$ holds, and there is no behavior protocol c such that c is a slice of b, $C(\mathscr{L}(c))$ holds, and $c \neq b$.

```
(
  !IDhcpCallback.IpAddressInvalidated*
  |
  (
    ?IManagement.UsePermanentIpDb↑ ;
    (
      !IIpMacPermDb.GetIpAddress*
      |
      (
        !IManagement.UsePermanentIpDb↓ ;
        ?IManagement.StopUsingPermanentIpDb↑
      )
    ) ;
    !IManagement.StopUsingPermanentIpDb↓
  );(
    ?IManagement.UsePermanentIpDb↑ ;
    (
      !IIpMacPermDb.GetIpAddress*
      |
      (
        !IManagement.UsePermanentIpDb↓ ;
        ?IManagement.StopUsingPermanentIpDb↑
      )
    ) ;
    !IManagement.StopUsingPermanentIpDb↓
  );(
    ?IManagement.UsePermanentIpDb↑ ;
    (
      !IIpMacPermDb.GetIpAddress*
      |
      (
        !IManagement.UsePermanentIpDb↓ ;
        ?IManagement.StopUsingPermanentIpDb↑
      )
    ) ;
    !IManagement.StopUsingPermanentIpDb↓
  )
)
```

Fig. 4. A minimal reduction of the DHCPServer frame protocol

3 Slicing with respect to composition

This section presents a concrete protocol slicing technique—slicing with respect to composition. This technique is the proposed solution addressing the goal articulated in Sect. 1.2, i.e. to develop a method to reduce behavior protocols with respect to their particular composition (reflecting a desired component architecture/assembly) in order to omit the parts of the behavioral specification superfluous with respect to the composition. The technique is based on the formal basis provided in Sect. 2, and the general slicing strategy described in [18], which aims at extending the program slicing paradigm to general slicing of an expression.

Again, the goal is to determine the unused behavior in a composition of given components and eliminate it from the behavior specification. Assuming the behavior of components is specified via their frame protocols FP_1, FP_2, … , FP_n and the language of the composition of the components is thus $LC = \mathscr{L}(FP_1 \nabla FP_2 \nabla \ldots \nabla FP_n)$, it is desirable to find minimal reductions $FP^{mr}{}_1$, $FP^{mr}{}_2$, … , $FP^{mr}{}_n$ of the protocols FP_1, FP_2, … , FP_n such that $\mathscr{L}(FP^{mr}{}_1 \nabla FP^{mr}{}_2 \nabla \ldots \nabla FP^{mr}{}_n) = LC$. In other words, the composition of the reduced protocols should specify precisely the same behavior as the composition of the original protocols. In terms of Def. 3, the predicate is chosen for each protocol FP_i separately as $C_i(\mathscr{L}(FP^{mr}{}_i)) \equiv (\mathscr{L}(FP_1 \nabla \ldots \nabla FP^{mr}{}_i \nabla \ldots \nabla FP_n) = LC)$. Moreover, it would be an advantage to echo this reduction in these frame protocols by their syntactical simplification—to slice them. However minimal reduction cannot be reflected accurately by slicing in general (Sect. 2.2); fortunately, as a compromise, minimal slicing of these frame protocols is achievable—see below. Thus instead of by a minimal reduction $FP^{mr}{}_i$, each FP_i is to be replaced by a minimal slice $FP^{ms}{}_i$ of it, again asking similarly $C_i(\mathscr{L}(FP^{ms}{}_i)) \equiv (\mathscr{L}(FP_1 \nabla \ldots \nabla FP^{ms}{}_i \nabla \ldots \nabla FP_n) = LC)$ to hold.

The general expression slicing strategy [18] prescribes slicing to be done in three phases: *parsing* – creation of syntax tree of an expression, *marking* – marking syntax tree nodes that are important with respect to the given slicing criterion, and *outputting* – creating slice of the original expression based on the marks on the syntax tree nodes.

Clearly, the actual logic of a specific slicing technique lies in the design of the phases, namely of the second one which determines how the slicing criterion is applied. The first and third phases are highly specific to the nature of the expression being sliced, but they do not influence the actual application of the slicing criterion.
The phases of the proposed slicing with respect to composition are as follows (Fig. 5). First, parsing of the protocols is done using the JavaCC [9] generated parser. The goal of the second phase is to mark the nodes of the syntax trees, which represent the behavior (sub)protocol relevant in the given composition. Basically, this is achieved by traversing the reachable states of the composition state space; all the syntax tree nodes that were used to generate reachable states are marked. This assures that the language of the resulting slice satisfies the constraint C_i articulated above. In the third phase, the slice is acquired by pruning the syntax trees of individual protocols based on the marks on their nodes (the unmarked nodes are removed).

By removing all the unmarked nodes, the technique clearly creates as small slices as possible, assuming that the protocols are not redundant, i.e. they do not specify redundantly (as e.g. protocol a? + a? does). This assumption does not cause any harm—real-life behavior protocols are usually not redundant, since redundancy does not introduce any new information into the behavioral specification.

For illustration, slicing of the behavior protocols ?a{!x}* | ?b* and !a{?x + ?y}* with respect to their composition via the consent operator ∇ is depicted in Fig. 5. In the 1st phase, parse trees of these behavior protocols are constructed. Then, in the 2nd phase, the behaviors specified by these protocols are composed via ∇. All the reachable states of the composition state space are sought on-the-fly and all the nodes of the parse trees that were used to generate the reachable states of the composition are marked. Finally, in the 3rd phase, the parse trees are pruned to contain only the marked nodes which make sense (note deletion of the | and + operators, when losing the second operand). The resulting sliced protocols are: ?a{!x}* and !a{?x}*, which are minimal slices (and even minimal reductions, in this special case).

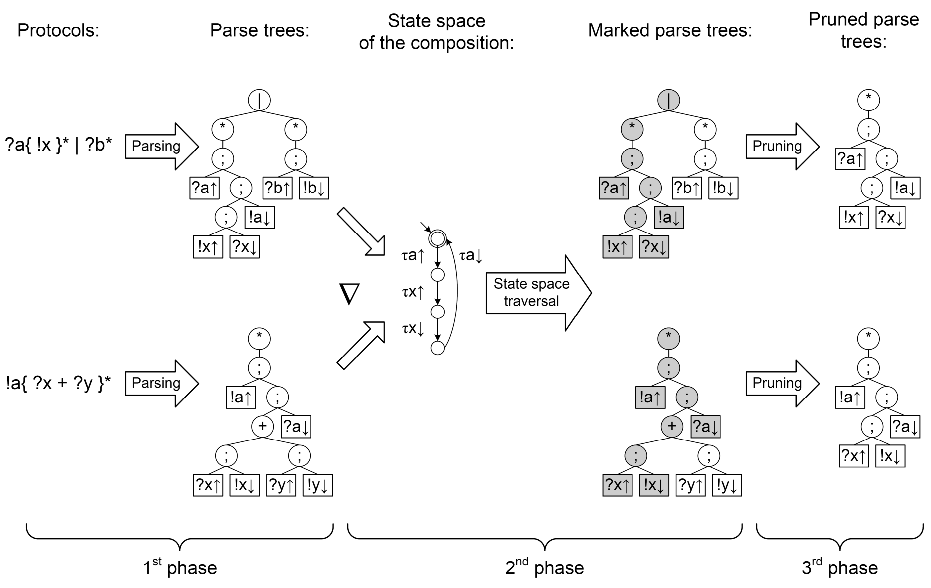

Fig. 5. Three phases of slicing with respect to composition

4 Tools and Case Study

The proposed technique was implemented as a stand alone application BPSlicer, being an extension of the dChecker behavior protocol checker written in Java 1.5, both available at [6].

BPSlicer was applied to the case study mentioned in Sect. 1 (Fig. 1), which was modified (reused) to provide internet access in a public garden, where credit card payment was considered as the only option. The frame protocol representing the environment was manually modified accordingly. Then the frame protocols of the components were sliced with respect to composition by BPSlicer. As expected, the frame protocols of `Token`, `Firewall`, `CardCenter`, and `AccountDatabase` remained unchanged, since they do not feature any airport specific functionality. On the other hand, the frame protocols of `FlyTicketDatabase` and `FrequentFlyerDatabase` were reduced to a NULL protocol, which means that these components were not used in the new environment and could be safely removed from the architecture. The frame protocols of `Arbitrator` and `DhcpServer` were reduced partially: The airport specific login calls were omitted from the frame protocol of `Arbitrator`. As to `DhcpServer`, the unused part describing the permanent MAC to IP address association was sliced off (actually this feature was not used even in the original application; note the unbound `IManagement` and `IIpMacPermDb` interfaces in Fig. 1).

Technically speaking, the slicing technique proposed in Sect. 3 is implemented as a part of the consent operator evaluation in dChecker which also uses parse trees and creates on the fly the state space of the parallel composition (BPSlicer adds the marking). As an aside, the state space generated by the behavior composition of the top-level components in Fig. 1 features around 4.5 millions of states and its error-free communication was verified by the dChecker in 126 seconds (Core Duo T2400 2x1.83 GHz, 1 GB RAM, 600 MB for Sun JVM 1.5.0.08, Linux 2.6.17). For comparison, the garden scenario, featured 421 980 states and took 23 seconds to verify the communication and slice the protocols to the size indicated in Table 1 (it took 18 seconds without slicing).

Table 1. Summary of the case study results. Reduced protocols are printed in bold script

protocols	*#states*	
	original	*reduced*
`Environment`	13	13
`Arbitrator`	15 625	**8 125**
`DhcpServer`	33	**3**
`Token`	245	245
`Firewall`	81	81
`AccountDatabase`	729	729
`CardCenter`	3	3
`FlyTicketDatabase`	7	**1**
`FrequentFlyerDatabase`	7	**1**

5 Related work

There are two main areas of related work. The first one includes research sharing our motivation—applying slicing to formal specification and/or the software architecture in order to facilitate its reuse and make its comprehension easier. In [8], Hassine *et al.* apply generalized slicing to functional requirement specification stated in Use Case Map notation. Their goal is to promote reuse of the requirement specification and aid with software maintenance by developing techniques that would help identify feature dependencies and interactions. Although the motivation is very similar to ours, the levels of abstraction differ.

The works by Stafford and Wolf [19] and Zhao [20] target slicing of software architecture description with similar goals. Stafford and Wolf provide the Aladdin tool for slicing of software architecture specified in Rapide [11]. In his work, Zhao describes a technique for reduction of software architecture in Wright [2]. Both these works aim at removing connectors and/or components from the software architecture based on the behavioral description and a slicing criterion. In contrast, our approach goes one-step further, because we can reduce unused behavior at a finer level of granularity—method calls, not being limited to granularity of components and connectors as in [19] and [20].

The second area pertains related work which is focused on component adaptation. In [16], Reussner presents a concept of *parameterized contract* on component interface. The contract specifies which of the provided interfaces of a component can be safely used if specific required interfaces are bound. This approach, in addition to a separate behavior specification on each interface, needs an explicit specification of the contract between the provided and required interfaces (in a different formalism). On the contrary, a frame protocol describes behavioral specification of a component as a whole (the interplay of calls on the proved and required interfaces), so that all the necessary information for adaptation of the specification is available in the protocol. In a similar vein, the relativity of a component's failure with respect to a particular environment it is used in is further discussed in [1].

Bobeff and Noye [4] use the techniques of program slicing and partial evaluation for component (code) specialization (adaptation). They envision delivery of generic components (*component generators)* that would automatically generate components adapted to the environment they are used in. When compared to our approach, [4] works at the code level, requiring it to be known at the time of adaptation. Our technique works solely on the level of behavioral specification and can be applied even when the actual code is not available, which is typical for COTS components. Similar to [4] is the Koala component framework [13], which statically optimizes the architecture for specific parameters. Our solution is more flexible since it takes into account also the behavior (not only static configuration of components).

6 Conclusion

In order to help a software designer with reusing software components and even whole component architectures, a technique for slicing behavior protocols, slicing with respect to composition, was presented. Given a composition of components, the technique can remove the unused behavior from the behavior specification, clarifying thus the actual roles of individual components.

Viability of the proposed technique was demonstrated by the prototype implementation and its use in a non-trivial case study in Sect. 4. Moreover, the formal foundation in Sect. 2 was designed to allow for an easy extensibility, so that it can be used as a basis for other slicing techniques than the one described. For example, slicing with respect to property, omitting the parts of the behavior specification irrelevant to a certain user-specified property, could be considered. We also envision the contribution of our work to the problem of modeling component environment for the purpose of code checking of isolated primitive components [14], as the presented technique of slicing with respect to composition can be used to restrict the model of a component's environment, reducing size of the state space to be explored and making code checking more feasible in this way.

Acknowledgements

The authors would like to thank to Pavel Ježek and Jan Kofroň for sharing the CRE demo application architecture diagram, being a basis for Fig. 1 and to Pavel Parízek for valuable comments and suggestions.

References

1. Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis, Journal of Software Maintenance and Evolution: Research and Practice 17(5), John Wiley & Sons, Inc., pp. 363–377, 2005.

2. Allen, R., Garlan, D.: A Formal Basis for Architectural Connection, ACM Transactions on Software Engineering and Methodology 6(3), ACM Press, pp. 213–249. 1997.

3. Bergstra, J.A., Ponse, A., Smolka, S.A.: Handbook of Process Algebra, Elsevier, ISBN: 0-444-82830-3, 2001.

4. Bobeff, G., Noye, J.: Molding Components using Program Specialization Techniques, Eighth International Workshop on Component-Oriented Programming, 2003.

5. The CRE project (Component Reliability Extensions for Fractal Component Model), http://kraken.cs.cas.cz/ft/public/public_index.phtml.

6. dChecker & BPSlicer, http://dsrg.mff.cuni.cz/projects/dchecker.

7. Gramlich, G., Schnitger, G.: Minimizing NFA's and Regular Expressions, In proceedings of the 22nd Annual Symposium on Theoretical Aspects of Computer Science, STACS 2005, LNCS 3404, Springer, pp. 399–411, 2005.

8. Hassine, J., Dssouli, R., Rilling, J.: Applying Reduction Techniques to Software Functional Requirement Specifications, In proceedings of the System Analysis and Modeling, 4th International SDL and MSCWorkshop, SAM 2004, LNCS 3319, Springer, pp. 138–153, 2004.

9. JavaCC (Java Compiler Compiler), https://javacc.dev.java.net.

10. Jiang, T., Ravikumar, B.: Minimal NFA problems are hard, SIAM Journal on Computing 22(1), pp. 1117–1141, 1993.

11. Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and Analysis of System Architecture Using Rapide, IEEE Transactions on Software Engineering 21(4), IEEE Press, pp. 336–355, 1995.

12. Meyer, A.R., Stockmeyer, L.J.: The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space, In proceedings of the 13th Annual Symposium on Switching and Automata Theory, FOCS, pp. 125–129, 1972.

13. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala Component Model for Consumer Electronics Software, IEEE Computer 33(3), pp. 78–85, 2000.

14. Parizek, P., Plasil, F.: Modeling Environment for Component Model Checking from Hierarchical Architecture, In proceedings of Formal Aspects of Component Software (FACS'06), Prague, Czech Republic, ENTCS, 2006

15. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering 28(11), IEEE Press, pp. 1056–1076, 2002.

16. Reussner, R.H.: Automatic component protocol adaptation with the CoConut/J tool suite, Tools for program development and analysis 19(5), Elsevier Science Publishers, pp. 627–639, 2003.

17. Sery, O.: Model Checking and Reduction of Behavior Protocols, Master thesis at Charles University in Prague, 2006, available at: http://dsrg.mff.cuni.cz.

18. Sloane, A.M., Holdsworth, J.: Beyond Traditional Program Slicing, In proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '96), ACM Press, pp. 180–186, 1996.

19. Stafford, J.A., Richardson, D.J., Wolf, A.L.: Architecture-level Dependence Analysis for Software Systems, International Journal of Software Engineering and Knowledge Engineering 11(4), pp. 431–451, 2001.

20. Zhao, J.: A Slicing-Based Approach to Extracting Reusable Software Architectures, In proceedings of the Conference on Software Maintenance and Reengineering, IEEE Computer Society, pp. 215–223, 2000.