

Model Checking of Component Behavior Specification: A Real Life Experience¹

Pavel Jezek, Jan Kofron, Frantisek Plasil

*Charles University in Prague
Department of Software Engineering
Czech Republic
{jezek, kofron, plasil} @ neny.ms.mff.cuni.cz
Academy of Sciences of the Czech Republic
Institute of Computer Science
Czech Republic
{kofron, plasil} @ cs.cas.cz*

Abstract

This paper is based on a real-life experience with behavior specification of a non-trivial component-based application. The experience is that model checking of such a specification yields very long error traces (providing counterexamples) in the order of magnitude of hundreds of states. Analyzing and interpreting such an error trace to localize and debug the actual specification is a tedious work. We present two techniques designed to address the problem: state space visualization and protocol annotation and share the positive experience with applying them, in terms of making the debugging process more efficient.

1 Introduction

1.1 Software Component Behavior and Model Checking

Model checking is one of the formal verification methods. Checking for important properties of a system (e.g. absence of deadlocks, array element indices within limits) assumes a model describing the system behavior is available. The model defines a state space and the desired property is verified via its

¹ This work was partially supported by the Czech Academy of Sciences project 1ET400300504, the Grant Agency of the Czech Republic project GACR 102/03/0672 and France Telecom under the external research contract number 46127110.

exhaustive traversal. In case of software model checking, a model can be obtained either from a system specification such as ADL (e.g. Wright [15], FSP [5], behavior protocols [1]) or via the source code analysis (the Bandera [10], SLAM [7] projects and Java PathFinder [11]).

Model checking faces two key inherent problems — state space explosion and error trace complexity and interpretation. An error trace is the path through the state space representing the particular computation in which the desired property is violated. The main problem regarding error traces is that a very long trace, in the order of magnitude of hundreds of states, may be very hard to analyze and interpret [21,22,23,24].

There are two widely used tactics for exhaustive traversal of the state space: Depth First Search (DFS) and Breadth First Search (BFS). Specification of a software unit (e.g. software component) usually generates huge state space. This is caused by the need of modeling large data type domains and parallelism (threads/processes). Therefore, the BFS-based tactics cannot be practically used because of their high memory requirements; instead, a DFS-based tactic has to be chosen. Unfortunately, in comparison with BFS, DFS has a drawback — the error trace it finds is not the shortest one in general.

1.2 Goals and Structure of the Paper

Behavior protocols [1] are a method of software component behavior specification. They are used for behavior specification in the SOFA [16] and the Fractal [4] component models. We employed behavior protocols in several non-trivial case studies of component behavior specification, comprising high number of components. This includes a non-trivial component-based test bed application in a project funded by France Telecom aiming at integration of behavior protocols into Fractal component model. One of the key lessons learned has been that the error trace length problem is severe and has to be addressed seriously. The goals of this paper are (i) to share with the reader the experience gained during specifying behavior of a non-trivial component-based application and show that the error trace length problem is really serious, and (ii) to describe the techniques we designed to address this problem.

These goals are reflected in the rest of the paper as follows: Sect. 2.1 and 2.2 shortly describe behavior protocols and Sect. 2.3 illustrates how to use them for component behavior specification and demonstrates the problem with the error trace length on a fragment of a non-trivial application that will be used as a running example. In Sect. 3, as the key contribution, the proposed techniques for addressing the error trace length and interpretation problems are described. Sect. 4 contains an evaluation of the proposed techniques while Sect. 5 discusses related work. Sect. 6 concludes the paper and suggests future research direction.

2 Behavior Protocol Checking

2.1 Behavior Protocols and Software Components

Software components are building blocks of software and communicate through interface bindings [4,15,16]. A component may provide some functionality by its *provides* (server) interfaces and may require other functionality from its environment (other components) through its *requires* (client) interfaces. As an example, consider the DhcpServer component on Fig. 3. It is a composite component built of two other components — ClientManager and DhcpListener that are bound via their Listener interfaces. The DhcpServer has a provides interface (Mgmt) and two requires interfaces (PermanentDb and Callback).

A *behavior protocol* [1] is an expression describing the behavior of a component; the *behavior* means the activity on component’s interfaces viewed as sequences (traces) of accepted and emitted method call events. A behavior protocol² is syntactically composed of event denotations (tokens), the operators (Fig. 1 and parentheses. For a method *m* on an interface *i*, there are four event token variants:

Emitting an invocation: $!i.m\uparrow$ Accepting an invocation: $?i.m\uparrow$
 Emitting a response: $!i.m\downarrow$ Accepting a response: $?i.m\downarrow$

Furthermore, three syntactic abbreviations of method calls are defined:

Issuing a method call: $!i.m$ is an abbreviation for $!i.m\uparrow;?i.m\downarrow$

Accepting a method call: $?i.m$ is an abbreviation for $?i.m\uparrow;!i.m\downarrow$

Processing of a method: $?i.m \{expr\}$ stands for $?i.m\uparrow;expr;!i.m\downarrow$ meaning that *expr* defines the *m*’s reaction to the call in terms of issuing and accepting other events.

Operator	Meaning
$;$	Sequence: $a;b$ means after <i>a</i> is performed <i>b</i> is performed
$+$	Alternative: $a+b$ means either <i>a</i> or <i>b</i> is performed
$*$	Repetition: a^* means <i>a</i> is performed zero to a finite number of times
$ $	And-parallel: $a b$ generates all arbitrary interleavings of the sequences defined by <i>a</i> and <i>b</i>
$ $	Or-parallel: $a b$ stands for $(a b) + a + b$

Fig. 1. Basic protocols operators

² In principle, behavior protocols are similar to CSP, however they are not defined via recursive equations, but by expressions only, and the generated traces are finite. Also, parallel operators $|$ and $||$ are syntactical abbreviation in principle (can be replaced by $+$ and $;$). Parallel composition in the sense of CSP is covered by the consent operator (Sect. 2.2). Since a full fledged definition of behavior protocols requires much more space than it is provided in this paper, we refer the reader for details to [1,2].

As an example consider the fragment of behavior protocol in Fig. 2. According to it, the ClientManager component is able to accept `RequestNew`, `Update` and `Return` method calls on the interface `Listener` in parallel any finite number of times. If a `Return` method call is accepted, the component reacts by performing a `Disconnected` method call on its `Callback` interface. Furthermore, a `Disconnected` method call can be emitted at any time.

```
(
  ?Listener.RequestNew
  ||
  ?Listener.Update
  ||
  ?Listener.Return { !Callback.Disconnected }
)* | !Callback.Disconnected*
```

Fig. 2. Fragment of the ClientManager frame protocol

Although a behavior protocol may define an infinite set of traces, each trace is finite — the repetition operator denotes any arbitrary finite number of its argument repetition. Each behavior protocol defines a finite automaton with transitions labeled by the protocol’s events.

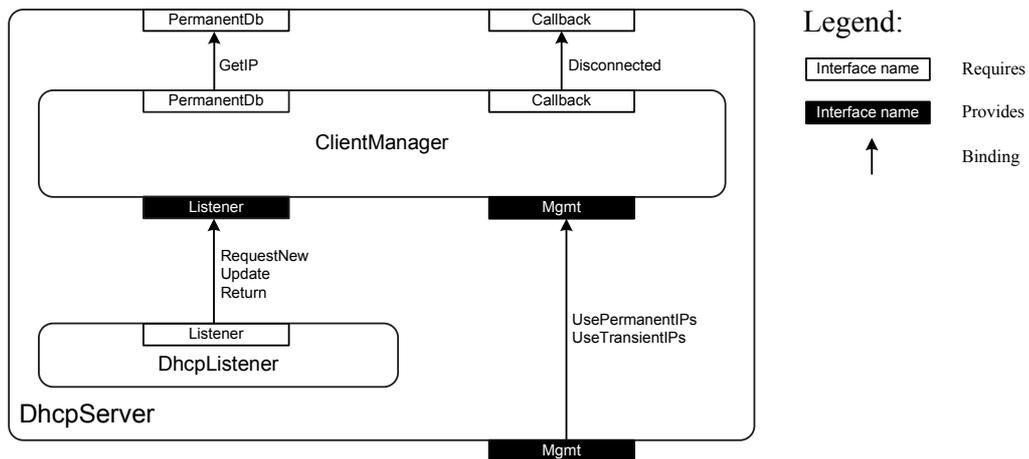


Fig. 3. DhcpServer composite component architecture

A *frame* (behavior) protocol of a component describes its "black-box" behavior (only the events on provides and requires interfaces are visible), while an *architecture* protocol of a (composite) component describes its behavior as defined by the composition of its first-level subcomponents, i.e. the communication events of these subcomponents appear in the behavior. Using the DhcpServer composite component in Fig. 3 as an example, its frame protocol contains only the events of the `Mgmt`, `PermanentDb` and `Callback` interfaces; the architecture protocol of the DhcpServer component is created by a parallel composition of frame protocols of DhcpListener and ClientManager components.

2.2 Protocol Compliance and Composition

The key benefit of using behavior protocols to describe behavior of components is at the design stage of an application. The developer can check whether the components he/she composes have compatible behavior: it enables for checking the component compatibility both horizontally (e.g. between the ClientManager and DhcpListener components) and vertically (between the DhcpServer frame protocol and the architecture protocol created by parallel composition of the ClientManager and DhcpListener frame protocols) [1].

The horizontal protocol compatibility is defined via the consent operator [2], which is basically a parallel composition converting the subcomponents' communication events to internal (τ) events. This is similar to CSP, however in addition the consent composition detects three kinds of composition errors: *bad activity*, *no activity*, and *infinite activity*. Bad activity occurs when a component emits a call on an interface and the component providing that interface is not able to accept (according to its behavior protocol) such a call. No activity is a deadlock and infinite activity means that there is "no agreement" in two composed repetitions on a joint exit (there is a loop that cannot be exited due to the nature of communication). The consent operator and composition errors are thoroughly described in [2].

The vertical compatibility is captured via protocol compliance [1]. The protocol compliance is defined between the frame protocol of a component and its architecture protocol, i.e. the protocol created from its subcomponents' frame protocols composed via the consent operator.

2.3 Example: A Fragment of the Test Bed Application

In this section we describe a fragment of a test bed application ("Wireless Internet Access") mentioned in Sect. 1.2. The application is a quite complex system allowing clients of various air-carriers to access the Internet from airport lounges via local Wi-Fi networks. The whole Wireless Internet Access application is composed of about 20 Fractal components. One of the key components is the DhcpServer composite component (Fig. 3). It communicates with system's clients at the lowest level, i.e. it is responsible for managing clients' IP addresses, monitoring overall state of the local wireless network and providing this information to the rest of the system. A simplified version is presented in this section.

2.3.1 DhcpServer Architecture

In principle, the DhcpServer composite component works in two functionality modes which can be swapped via the `Mgmt` interface:

- (i) *DhcpServer generates IP addresses dynamically* for new clients (this is the default functionality that can be also set by calling the `UseTransientIPs`

method on the `Mgmt` interface).

- (ii) *DhcpServer assigns IP addresses statically* based on mappings between clients' MAC and IP addresses in an external database accessible via the `PermanentDb` interface (this functionality is set by calling the `UsePermanentIPs` method on the `Mgmt` interface).

When a client disconnects from the network, the `DhcpServer` calls the `Disconnected` method on its `Callback` interface to notify its environment about this event.

As already mentioned, the `DhcpServer` functionality is implemented by its subcomponents: `ClientManager` and `DhcpListener`. The architecture of the `DhcpServer` and bindings between the subcomponents is shown on Fig. 3.

```
(
    !Listener.RequestNew
    ||
    !Listener.Update
    ||
    !Listener.Return
)*
```

Fig. 4. Frame protocol of `DhcpListener`

The `DhcpListener` component is responsible for the "real" communication with network clients and the network infrastructure. Internally it uses existing system infrastructure to manage client nodes. Events that occur at the network level are unified by `DhcpListener` which converts them to method calls. As they can arrive at any time, the corresponding frame protocol has to express the inherent parallelism (Fig. 4).

`ClientManager` accepts notifications on network events from the `DhcpListener` and processes them either internally (`RequestNew` and `Update`) or forwards them to `DhcpServer`'s environment (via `Callback.Disconnected`) as part of `Return` processing.

`ClientManager`'s behavior is expressed by its frame protocol in Fig. 5. The part A of the protocol represents the "generate IP addresses dynamically" functionality of `ClientManager` while the part B represents the "assign IP addresses statically" functionality. The parts A.1 and B.1 express the `ClientManager`'s ability to process `DhcpListener`'s notifications and also describe reactions to them. The parts A.2 and B.2 capture `ClientManager`'s ability to detect client disconnections internally, resulting in a call of `Disconnected`. The `ClientManager`'s functionality mode swapping mechanism is reflected in the parts A.3 and B.3: At any time, `ClientManager` can accept a method call requesting a mode change (`?Mgmt.UsePermanentIPs↑` or `?Mgmt.UseTransientIPs↑`), but it does not respond it immediately. Instead, it waits until the processing of all pending method calls on the `Listener` interface is finished and then it issues the `!Mgmt.UsePermanentIPs↓` or the `!Mgmt.UseTransientIPs↓` response. Then `ClientManager` is again ready to accept further calls on the

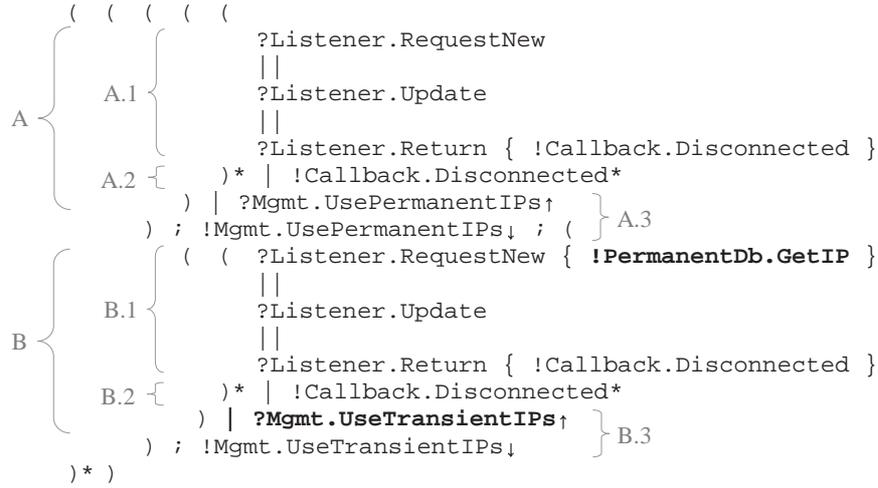


Fig. 5. Frame protocol of ClientManager (The highlighted lines denote the events forming the composition error described in Sect. 2.3.3)

Listener interface and respond to them according to its newly set functionality mode.

2.3.2 DhcpServer Frame Protocol

The frame protocol of DhcpServer is shown in Fig. 6. The interactions between DhcpServer’s subcomponents are not visible in it. However, their communication can trigger interaction with the environment of DhcpServer that is therefore visible in its frame protocol. This is illustrated by the part C of the frame protocol in Fig. 6: the **!Callback.Disconnected** call can be invoked by the ClientManager subcomponent either as a reaction to an accepted **?Listener.Return** call or due to its internal detection of client disconnection (Sect. 2.3.1); however these two causes are indistinguishable in the DhcpServer frame protocol. The part D of the protocol expresses the DhcpServer’s ability to swap between its two modes (Sect. 2.3.1).

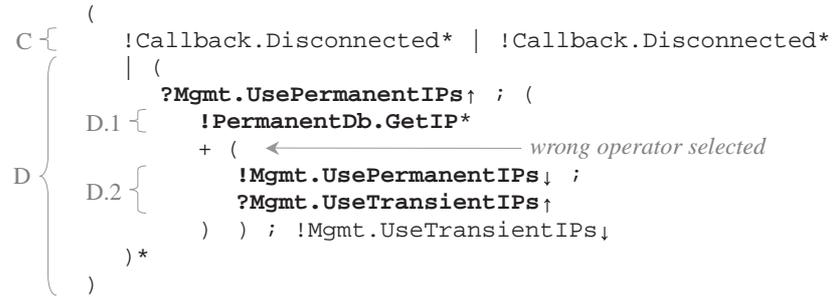


Fig. 6. First version of the frame protocol of DhcpServer (Instead of +, the | operator should have been used here as demonstrated by the error trace in Sect. 2.3.3)

2.3.3 Checking for Composition Errors and Compliance

The application developer that sets up a composite component (such as DhcP-Server) creates also its frame protocol, whereas the frame protocols of sub-components (ClientManager and DhcPListener) are created by their respective authors.

It is the developer’s responsibility to check first for composition errors (horizontal compatibility) between subcomponents (Sect. 2.2). The frame protocols of ClientManager and DhcPListener (Sect. 2.3.1) as presented above are compatible in this sense. It should be emphasized that behavior incompatibility may occur even though the components are connected via type-compatible interfaces.

The next step in a composite component’s development is to check for compliance (vertical compatibility (Sect. 2.2)) of its frame protocol with its architecture protocol. During the development of the first version of the DhcP-Server component, the + operator was used in its frame protocol (Fig. 6). However, such a protocol was not compliant with its architecture protocol (Sect. 2.3.1). Using the behavior protocol checker, the error was found and reported by an error trace (Fig. 7).

(S0) <code>τListener.Return↑</code>	(S117) <code>τListener.RequestNew↓</code>
(S1) <code>τListener.Update↑</code>	(S118) <code>τListener.Update↑</code>
(S2) <code>τListener.Update↓</code>	(S127) <code>τListener.Update↓</code>
(S3) <code>τListener.RequestNew↑</code>	(S128) <code>τListener.Return↑</code>
(S4) <code>τListener.RequestNew↓</code>	(S129) <code>τCallback.Disconnected↑</code>
(S5) <code>τMgmt.UsePermanentIPs↑</code>	(S130) <code>τCallback.Disconnected↑</code>
(S6) <code>τCallback.Disconnected↑</code>	(S171) <code>τCallback.Disconnected↓</code>
(S7) <code>τCallback.Disconnected↑</code>	(S188) <code>τListener.Return↓</code>
(S46) <code>τCallback.Disconnected↓</code>	(S189) <code>τListener.Update↑</code>
(S47) <code>τListener.Return↓</code>	(S190) <code>τListener.Update↓</code>
(S48) <code>τListener.Return↑</code>	(S191) <code>τListener.RequestNew↑</code>
(S49) <code>τListener.Update↑</code>	(S192) <code>τListener.RequestNew↓</code>
(S50) <code>τListener.Update↓</code>	(S193) <code>τCallback.Disconnected↓</code>
(S51) <code>τListener.RequestNew↑</code>	(S226) <code>τMgmt.UsePermanentIPs↓</code>
(S52) <code>τListener.RequestNew↓</code>	(S227) <code>τListener.Return↑</code>
(S53) <code>τCallback.Disconnected↓</code>	(S228) <code>τListener.Update↑</code>
(S54) <code>τCallback.Disconnected↑</code>	(S229) <code>τListener.Update↓</code>
(S55) <code>τCallback.Disconnected↓</code>	(S230) <code>τListener.RequestNew↑</code>
(S56) <code>τListener.Return↓</code>	(S231) <code>!PermanentDb.GetIP↑</code>
(S57) <code>τListener.RequestNew↑</code>	

Fig. 7. Error trace representing a compliance error

However, identifying the actual error only from such a plain error trace is not a trivial task. The key problem is that error traces of real components tend to be rather cryptic; in particular, several method calls of the frame protocol can occur in parallel. This leads to interleaving of the error-related events with other events processed in "background". For example, only the highlighted events on Fig. 7 lead to the conclusion that the parts D.1 and D.2 of DhcP-Server’s frame protocol (Fig. 6) need to be processed in parallel,

because the `ClientManager` can issue the `!PermanentDb.GetIP` call (in B.1) in parallel with accepting the `?Mgmt.UseTransientIPs↓` call (in B.3).

3 Approaches to Error Trace Analysis and Interpretation

In behavior protocols, an error trace's end is reflected in the state space (defined by the protocol) as a state F . It is a specific feature of behavior protocols that each trace reaching F is an error trace. Hence, F is an *error state*. In consequence, an error state represents a set of error traces SF . (Note that the existence of error states is not a general feature of an LTS.) Finding all elements of SF means complete traverse of the state space. Sometimes, however, the knowledge of the whole set of error traces corresponding to an error state may be very beneficial for error cause's identification. As the set of error traces may be huge (or even infinite), providing it as a list of traces would not be of much help. Therefore, additional forms of SF representation are needed.

3.1 Plain Error Trace

As demonstrated in Sect. 2.3.3, an error trace identifying a compliance or composition error may be quite long and hard to interpret. Moreover, due to the DFS tactic used, the error trace may contain states not capturing "the essence" of the error. For example, the state subsequence $S5, S226, S230, S231$ of the error trace in Fig. 7 also forms an error trace, but the longer one was found first. In this respect, the other states are "not-important" ones. It is a challenge to filter out these "not-important" states (to find a canonical representation of the error trace set associated with an error state). One can imagine a filtering technique based on iterative re-searching the state space, which would take advantage of the knowledge of the depth at which the error was found.

3.2 State Space Visualization

One of the checking outputs we propose in order to make error interpretation easier is *state space visualization*. Visualization is a graphical representation of the state space associated with the protocol (Sect. 2.2). For the state space related to Sect. 2.3.1, this is illustrated on Fig. 8 (only a fragment of the state space is captured here for brevity). This helps find out what the problem cause is by tracking the error trace in the state space.

Apparently, state space size might be a problem here — a state space having more than 1,000 states is hard to visualize. Thus, visualizing only a part of the state space becomes a practical necessity. In this perspective, capturing only the part containing the error state and its "neighborhood"

is a straightforward thought. We employed this idea with a very positive experience. Such a result still provides useful information, detailed enough to identify where the essence of an error is. Technically, our visualization outputs all the transitions leading from a state on the error trace — this helps with finding correspondence with the original protocol.

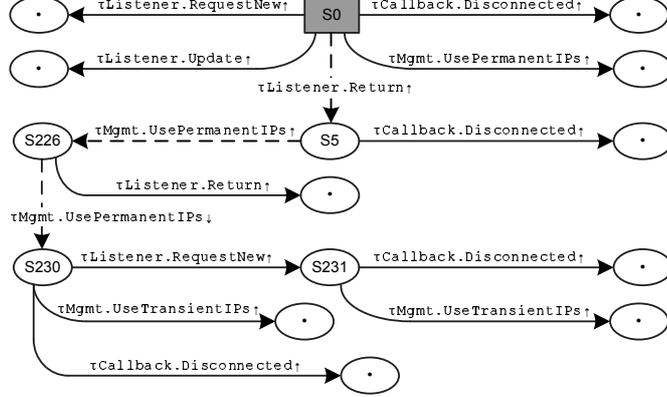


Fig. 8. State space visualization — dashed lines represent longer paths omitted due to the limited space of this paper. The state S231 is the error state F.

3.3 Protocol Annotation

Another way of representing an error state are *annotated protocols*. Consider a composition of protocols P and Q via the consent operator. If the composition yields a composition error in an error state S, the state S is represented by marks <HERE> put into P and Q, forming the annotated protocols PS and QS. For illustration consider Fig. 9 where a fragment of the annotated frame protocol of DhcpServer corresponding to the error trace in Sect. 2.3.3 is depicted.

Advantageously, there is no need to construct the entire state space, but it suffices to annotate only the protocols featuring as operands in a composition. For example, the set of error traces specified by the annotated protocol in Fig. 9, together with the annotated architecture protocol of DhcpServer internals, yields the error traces:

```

 $\tau\text{Callback.Disconnected}\uparrow$ ;  $\tau\text{Callback.Disconnected}\downarrow$ ;
 $\tau\text{Mgmt.UsePermanentIps}\uparrow$ ;  $\tau\text{Mgmt.UsePermanentIps}\downarrow$ 
and
 $\tau\text{Mgmt.UsePermanentIps}\uparrow$ ;  $\tau\text{Mgmt.UsePermanentIps}\downarrow$ ;
 $\tau\text{Callback.Disconnected}\uparrow$ ;  $\tau\text{Callback.Disconnected}\downarrow$ 

```

There are two issues to be addressed with this technique:

- (i) *Identical prefixes in alternatives.* For example, consider the following

frame protocol: $(?i.m1; ?i.m2) + (?i.m1; ?i.m3)$. If an error state is to be indicated after $?i.m1$, the corresponding annotated protocol takes the form:

$$(?i.m1<HERE>; ?i.m2) + (?i.m1<HERE>; ?i.m3)$$

Even though one of the alternatives could be eliminated, we prefer keep them both to provide more context of the error.

- (ii) *Transformations performed on input protocols.* In the protocol checker, the protocols are modified during the parsing process (e.g. $?i.m$ is decomposed into $?i.m\uparrow; !i.m\downarrow$ and the formatting information is lost). Therefore, exact mapping of an error state back to the source protocols may be difficult. Fortunately, the transformations typically still yield a reasonably readable behavior protocol, which, annotated, provides useful information for specification debugging.

```
( (
  ?Callback.Disconnected↑; !Callback.Disconnected↓<HERE>
)* ) | ( (
  !Mgmt.UsePermanentIps↑ ; (
    (?PermanentDb.GetIp↑; !PermanentDb.GetIp↓)*
  ) + (
    ?Mgmt.UsePermanentIps↓<HERE>;
    !Mgmt.UseTransientIps↑
  ) ; ?Mgmt.UseTransientIps↓*
) )
```

Fig. 9. DhcpServer annotated frame protocol - simplified.

4 Evaluation

During the work on the case study mentioned in Sect. 2.3, it has turned out that combining all of the three forms of checking output is the most promising approach. Even though protocol annotation (Sect. 3.3) appears a very generic technique, in complex cases the other checking outputs have to be also provided, since tracking all the path alternatives in a annotated complex protocol may be error-prone.

The most complex components of the case study have behavior protocols with up to 60 events; such behavior protocols generate a state space with hundreds of thousands of states. The typical errors encountered during the development of such components then generate error traces of about 100 states in length. However there were also some error states that generated error traces with several hundreds of states. It then took the developer about an hour (often even more) to identify the actual error in case only a plain error trace was available. The checking output techniques presented in Sect. 3 have been developed to improve debugging efficiency. During the further development of

our case study application, the developers used a combination of these techniques and an average time to resolve a typical error shortened down to one third or one fourth of the original time.

As for the plain error trace checking output, a problem is the existence of "local loops" in behavior of a component. Typically, with respect to the other parts of the system, the actual number of local loop traversals is of no significance in terms of an error localization. These loops lengthen the error trace, making it more complex and hard to analyze. Apparently, if loops are nested, the situation is even worse. A desire is to eliminate those of "no influence" on the rest of the system. This is a challenging problem - currently, only the highest-level loops are identified and eliminated in an automated way.

Annotated protocols are very similar to the approach used in Bandera Toolset [10] and PREfast [3] since they are based on emphasizing of the positions in the input protocols where a composition error has been found. Unlike in Bandera and PREfast, in behavior protocols the positions between two operations are highlighted to denote an error state.

5 Related Work

In [23], the authors address the counterexample complexity and interpretation problem by proposing a method for finding "positives" and "negatives" as sets of related correct traces and error traces. An interesting approach is chosen in [21], where the authors analyze the complexity of error explanation via constructing the "closest" correct trace to a specific error trace. In [24], the authors describe an algorithm ("delta debugging") for finding a minimal test case identifying an error in a program. This idea could be used to modify an error trace in order to find a "close enough" correct one. An optimization of the checking process is described in [22] where multiple error traces are generated in a single checking run.

Static Driver Verifier (SDV) [6] is a tool used to verify correct behavior of WDM (Windows Driver Model) [8] drivers. The driver's source code in C and the model written in SLIC (a part of the SLAM project [7]) are combined into a "boolean" program that is maximally simplified and selected rules are checked. If a rule is violated, an error trace of the program is generated and mapped back to the driver's C source code. Because WDM drivers are very complex, to make checking feasible, both the Windows kernel model and the rules used in the SDV have to be simplified. Thus the error traces generated by SDV are relatively short and easy to interpret. And, since they contain also the states corresponding to traversing through the kernel model, such parts are optionally hidden in the checking output. This solution might be also applicable to our plain error traces (Sect. 3.1): The events generated inside a method call could be grouped into the "background" (Sect. 2.3.3).

However, because it is not easy to identify the beginning and the end of a single method call in error trace (especially when the `i.m{...}` shortcuts are not used), employing this idea in the behavior protocol checker is not a trivial task.

As to the classical model checker SPIN [9], in case of violating of checking property specified in LTL, Spin allows traversing the trace to the error state while watching the variable values, process communication graph, and highlighted source code. Sometimes the error trace length makes this approach very hard to use and identification of the actual problem may be quite challenging. Although the approaches to ease the interpretation of an error trace in SPIN work well in most cases, its modelling language Promela [9] is not a suitable specifying software components. Since such specification in Promela typically yields a large state space impossible to traverse in a reasonable time.

As for other tools, Java PathFinder (JPF) [11], Bogor [17], BLAST [18], SMV [12], Moped [19], and MAGIC [20] cope with counterexamples and all provide them as error traces. Specifically, JPF, Bogor, BLAST, Moped, and MAGIC print the sequence of steps leading to an error state annotated by a corresponding line of the source code, while the SMV tool provides an error trace consisting of the input file lines written in the SMV specification language. Moped is similar to SDV in the sense that it first translates the input program (in Java) into the language of LTL in which the counterexamples are generated. They are then translated back to the input language. The MAGIC tool checks behavior of a C program against a specification described via an LTS. Besides an error trace, it can also generate control flow graphs and LTSs using the dot tool of GraphViz package [13] (also used by the behavior protocol checker). In all cases, but especially in the case of JPF, the error trace may get quite complex and not easy to interpret.

6 Conclusion and Future Work

During the work on the project (Sect. 2.3.1) it has turned out that, besides plain error trace, additional checking outputs are needed for speeding up error detecting and debugging process. Therefore, we introduced two more approaches: (i) state space visualization, and (ii) annotated protocols. Using all the three methods in combination was found most beneficial (locating an error was then more efficient (Sect. 4)).

Problems arise when checking the composition/compliance of several components described by really complex behavior protocols. The large state space generated by such a protocol causes that an error trace is typically very long and hard to interpret. Still, in our view, this is worth to pursue since we believe that the components' compatibility problem cannot be restricted to the syntactic/type compatibility of their (bounded) interfaces [1], even though

this could be checked with much smaller effort and would avoid the problems discussed in this paper; in fact, we can hardly imagine putting together a non-trivial component-based application of the size mentioned in Sect. 2.3.1, if the compliance checks were based only on syntactic/type compatibility of individual interfaces.

Our future work is therefore focused on improving the methods currently used by the behavior protocol checker; in particular, a method for automated removing of unnecessary "local loops" (Sect. 4) would further simplify the plain error trace checking output.

As for state space visualization, an automated method for detecting the "important" part of the state space (currently done by hand) is needed to simplify the resulting graphical representation of an error trace.

Similar to Bandera [10] and PREfast [3], the possibility to dynamically indicate the correspondence between a particular position in an error trace and the associated part of the protocol would perhaps further ease and speed up the debugging process.

References

- [1] F. Plasil, S. Visnovsky, "Behavior Protocols for Software Components," IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002
- [2] J. Adamek, F. Plasil, "Component Composition Errors and Update Atomicity: Static Analysis," Journal of Software Maintenance and Evolution: Research and Practice, vol. 17, no. 4, John Wiley, 2005
- [3] PREfast — <http://www.microsoft.com/whdc/devtools/tools/PREfast.aspx>
- [4] E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, J-B. Stefani. An Open Component Model and Its Support in Java. 7th SIGSOFT International Symposium on Component-Based Software Engineering (CBSE7), LNCS 3054, Edinburgh, Scotland, May 2004.
- [5] J. Magee, J. Kramer, "Concurrency: State models & Java programs," John Wiley & Sons Ltd, ISBN 0-471-98710-7, 1999
- [6] SDV — <http://www.microsoft.com/whdc/devtools/tools/SDV.aspx>
- [7] T. Ball, S. K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis," POPL 2002, ACM, Jan 2002
- [8] WDM — <http://www.microsoft.com/whdc/archive/wdm.aspx>
- [9] Spin, Promela — <http://spinroot.com/spin>
- [10] Bandera — <http://bandera.projects.cis.ksu.edu>

- [11] Java Pathfinder — <http://javapathfinder.sourceforge.net>
- [12] SMV — <http://www-2.cs.cmu.edu/~modelcheck/smv.html>
- [13] GraphViz — <http://www.research.att.com/sw/tools/graphviz>
- [14] Mach, M., Plasil, F., Kofron, J., “Behavior Protocol Verification: Fighting State Explosion,” International Journal of Computer and Information Science, ACIS vol. 6, no. 1, Mar 2005
- [15] Wright — <http://www-2.cs.cmu.edu/~able/wright>
- [16] SOFA — <http://sofa.objectweb.org>
- [17] Bogor — <http://bogor.projects.cis.ksu.edu>
- [18] BLAST — <http://www-cad.eecs.berkeley.edu/~blast>
- [19] Moped — <http://www.fmi.uni-stuttgart.de/szs/tools/moped>
- [20] MAGIC — <http://www-2.cs.cmu.edu/~chaki/magic>
- [21] N. Kumar, V. Kumar, M. Viswanathan, “On the Complexity of Error Explanation,” VMCAI’05, ACM, 2005
- [22] Ball, T., Naik, M., Rajamani, S., “From symptom to cause: Localizing errors in counterexample traces,” Proceedings of POPL 2003, ACM, 2003
- [23] Groce, A., Visser, W., “What went wrong: Explaining counterexamples,” Proceedings of the SPIN Workshop on Model Checking of Software, LNCS 2648, Springer, 2003
- [24] Zeller, A., “Isolating cause-effect chains for computer programs,” Proceedings of FSE 2002, ACM 2002