Table of Contents

Exceptions in Component Interaction Protocols - necessity 1
Frantisek Plasil and Viliam Holub
1 Introduction 1
2 Background - behavior protocols 2
3 Handling exceptions in behavior protocols 4
3.1 Primitive techniques 4
3.2 Analyzing the problem and sketching a solution 7
3.3 Proposed solution - details 10
4 Case study 11
5 Evaluation 14
6 Related work 15
7 Conclusion 16

Exceptions in Component Interaction Protocols - necessity*

Frantisek Plasil[1,2] and Viliam Holub[1]

[1] Charles University, Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{plasil,holub}@nenya.ms.mff.cuni.cz,
WWW home page: http://nenya.ms.mff.cuni.cz/
WWW home page: http://nenya.ms.mff.cuni.cz/~holub/
WWW home page: http://nenya.ms.mff.cuni.cz/~plasil/
[2] Academy of Sciences of the Czech Republic
Institute of Computer Science
plasil@cs.cas.cz
WWW home page: http://www.cs.cas.cz/

Abstract. At ADL level, most of the current interaction protocols designed to specify components' behavior at their interfaces do not allow to capture exceptions explicitly. Based on our experience with real-life component based applications, handling exceptions as first class entities in a (formal) behavior specification is an absolute necessity. Otherwise, due to the need to capture exceptions indirectly, the specification becomes very complex, therefore hard to read and, consequently, error-prone. After analyzing potential approaches to introducing exceptions to LTS-based interaction specification (expressed via terms/expressions) in ADL, the paper presents the way we built exceptions into the behavior protocols. Finally, we discuss the positive experience with applying these exception-aware behavior protocols to a real-life Fractal component model application.

1 Introduction

There are many approaches to describe the desired behavior of software components. They include interface automata[4], behavior protocols[16], DFSM[19], usage policies[6], interactions and reactions[25], parametric contracts[18], UML2.0 State Machines (in principle stemming from Harel diagrams[8]) and Protocol State Machines[12], and CSP-based mechanisms, such as Wright[5] and FSP[11].

Those of them which are based on LTS (Label Transition System) where the transitions model atomic actions, allow for some kind of reasoning on behavior (e.g. equivalence[5], compatibility[4], compliance[16]). For instance, these atomic actions model the request and response triggered by a method call -

* This work was partially supported by the Grant Agency of the Czech Republic project 201/06/0770; the results will be used in the OSIRIS/ITEA project.

i.e. mostly the "control-observing" behavior of a component. Obviously, those LTS-based behavior description mechanisms which are directly applicable in architecture description languages cannot explicitly utilize any kind of diagrams, and therefore typically employ some kind of term expressions. However, there is a problem with this approach: capturing exceptions. While in a graphically expressed transition system, an exception can be expressed by adding another transitional edge, most of the term-expression based formalisms do not allow this easily. We encountered the problem when we were trying to employ behavior protocols [1,2,3,16], in non-trivial case studies of component behavior specification, comprising over 20 components each.

This paper aims at achieving two main goals:

1. To present a "reasonable" syntax extension of behavior protocols which does not violate the inherent regularity of the traces generated by the protocol (and therefore preserves important properties like protocol compliance decidability).
2. To show that the proposed syntax increases readability and significantly simplifies a behavior protocol when an exception is to be thrown/handled. This claim is supported by experimental results.

The paper is structured as follows: Sect. 2 describes the background - behavior protocols, in Sect. 3, the problem of handling exceptions in protocols is analyzed from a perspective of component communication and a solution is proposed. Section 4 illustrates the proposed solution on a case study. Section 5, as a part of overall evaluation, shares with the reader the experience with applying the proposed approach to a real-life Java project. Finally, Sect. 6 is focused on related work and Sect. 7 draws a conclusion.

2 Background - behavior protocols

The basic idea of behavior protocols can be illustrated on the following example (Fig. 1).

The picture shows the internal structure of a hypothetical Reservation component which is composed of five sub-components - Ticket manager (responsible for registration of tickets), Database manager (implementing the database behavior), Storage (permanently stores data), VISA (for payment authentication) and Operator verification (a connection to third-party servers).

Via behavior protocols, we can capture communication among these components. There are three types of protocols - *frame protocol* specifying the expected activities on components' boundaries (their frame), *architecture protocol* created automatically as a parallel composition of the frame protocols of the subcomponents (at the first-level of nesting) and the *interface protocol* describing the behavior only on a selected interface.

These abstractions allow for addressing two aspects of "design by contract": (i) *Horizontal contract* "Do the children cooperate with no conflicts?" (a conflict is statically detected as a *composition error*), and (ii) *vertical contract* "Do the

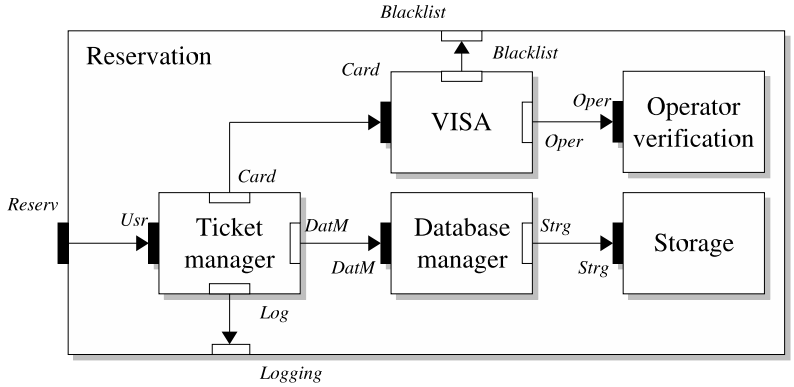

Fig. 1. The Reservation component

cooperating children do what the parent expects?" which is statically verified via evaluating *compliance* of the architecture protocol (determined by the subcomponents) and the frame protocol of the parent component. As an aside, the static composition error detection and compliance verification is done by a tool, protocol checker, available as a part of the SOFA project[21].

As to (i), three types of composition errors are identified - *bad activity* (an emitted event is not accepted), *no activity* (deadlock), and *divergence* (infinite activity). Definition of the semantics of compliance is crucial and has evolved from a *naive*[17] and *pragmatic*[16], to *consensual*[2] based on the idea that the architecture should work well (without composition errors) when cooperating with a separate component representing the architecture's environment. The behavior of this environment is defined as the "inverted" frame protocol of the parent component[2].

For example, compliance of the architecture protocol specifying the composed behavior of Ticket manager, Database manager, Storage, VISA and Operator verification with the frame protocol of Reservation can be verified. The frame protocol of the Ticket manager component can take the form:

```
?Usr.init;
(
  ?Usr.buyTicket {
    !DatM.preReserve;
    !Card.lookUp; !Card.payment;
    !DatM.reserve; !DatM.commit;
    !Log.print}
  +
  ?Usr.returnTicket {
    !Card.revert;
    !DatM.free; !DatM.commit}
)*;
?Usr.finish
```

The frame protocol specifies that the component expects (`?`) an `init` method call on the interface `Usr` followed (`;`) by alternatively (`+`) a call of `buyTicket` or `returnTicket`. After this is repeated a finite number of times (`*`), a `finish` call is accepted; no other incoming calls are allowed. The statements of the form `?i.a{P}`, where `P` is a subprotocol, `i` is an interface name and `a` is a method name, is an abbreviation of `?i.a↑;(P)!i.a↓`. The `?i.a↑` means *accepting* (`?`) a *request* (↑) `i.a`, and the `!i.a↓` means *emitting* (`!`) a *response* (↓) to `i.a`. Along these lines we see that the `buyTicket` method (acquiring and reserving a ticket for the user), calls (`!`) the `preReserve` on the `DatM` interface to inform the Database manager that a ticket is to be reserved. As a next step, calls of `lookUp` and `payment` methods on the `Card` interface are made to perform the payment; further, the `reserve` and `commit` methods on the `DatM` interface are called in order to confirm the transaction. The last action is to print information about the transaction - `Log.print`.

Instead of the alternative operator `+`, we could use the *or-parallel operator* `||` to express that calls of `buyTicket` and `returnTicket` might be accepted simultaneously. Additional operators and further details are described in [16].

3 Handling exceptions in behavior protocols

3.1 Primitive techniques

In real settings, exceptional situations (not described in the previous protocol) also have to be handled - e.g. the VISA component may deny service due to a network error, and the Database manager may refuse to allocate appropriate resources. In other words, specifying behavior of a component inherently involves exceptions. However, expressing exception via the standard operators is tedious. For illustration, consider the `DatM.preReserve` method call from the example above which could throw a `preReserveException` exception. In order to specify this behavior, we have to split the return from the `preReserve` method into a regular return (`?DatM.preReserve↓`) and an accepting return with exception (`?DatM.preReserveException↓`) - we call this technique *intrinsic exceptions* handling. However, in consequence, the frame protocol length would expand rapidly (exponentially in the number of methods throwing an exception).

Below is a fragment of the frame protocol of Ticket Manager where several exceptions are thrown and handled - it illustrates how the protocol becomes complex. In the example, we suppose that the `DatM.preReserve` method can throw `PreReserveException`, methods `Card.lookUp` and `Card.payment` can throw `NetworkException`, and finally the `DatM.reserve` method can throw `ReservationException`.

Technique: Intrinsic exceptions

```
...
?Usr.buyTicket↑;
  !DatM.preReserve↑;
  ( ?DatM.preReserve↓; !Card.lookUp↑;
    ( ?Card.lookUp↓; !Card.payment↑;
      ( ?Card.payment↓; !DatM.reserve↑;
        ( ?DatM.reserve↓; !DatM.commit↑;
          ( ?DatM.commit↓; !Usr.buyTicket↓ )
          +
          (  // exceptions of DatM.commit
            ?DatM.DatabaseException;
            !DatM.cancel; !Card.revert; !Log.print;
            !Usr.buyTicket↓
          )
        )
        +
        (  // exceptions of DatM.reserve
          (?DatM.DatabaseException↓+?DatM.ReservationException↓);
          !DatM.cancel; !Card.revert; !Log.exEvent; !Log.print;
          !Usr.buyTicket↓
        )
      )
      +
      (  // exceptions of Card.payment
        ?Card.NetworkException↓;
        !DatM.cancel; !Card.revert;
        !Usr.NetworkException↓
      )
    )
    +
    (  // exceptions of Card.lookUp
      ?Card.NetworkException↓;
      !DatM.cancel; !Card.revert;
      !Usr.NetworkException↓
    )
  )
  +
  (  // exceptions of DatM.preReserve
    ?DatM.PreReservationException↓; !Log.exEvent; !Log.print;
    !Usr.buyTicket↓
  )
...
```

Obviously, a part of the complexity of the problem is the fact that we have to separate requests and responses of method calls to capture that exceptions can happen between them. Moreover the "reaction" inside such a call has to be divided into a "regular" and an exception part, and, even worth, the exception part has to contain repeatedly the "regular" continuation of the method (notice how

many times is `!Log.print` appears in the specification). Clearly, if we could take advantage of keeping the expressive power of the abbreviations `?a{P}` or `!a{P}`, and add specific syntactical constructs for capturing exceptions as classical programming languages do, we could shorten this behavior protocol significantly and make it much more concise and, consequently, easier to comprehend.

Another option is to use the *approximation by alternative* technique the basic idea of which is to put after any method call alternative, non-deterministically chosen reactions (`+`) covering all the potential continuations. These include "regular" continuation, and those specific for each of the exceptions the method can throw. An example of this technique is below. For instance, `!DatM.reserve` is followed by alternatively calling `!DatM.commit` or handling the reservation exception (the `!DatM.cancel; !Card.revert; !Log.exEvent; !Log.print` part). Obviously, this approach only approximates real behavior of a component by not explicitly specifying the issuing and accepting events related to an exception.

Technique: Approximation by alternative

```
...
Usr.buyTicket {
  !DatM.preReserve;
  ( !Card.lookUp;
    ( !Card.payment;
      ( !DatM.reserve;
        ( !DatM.commit;
          (
            null +
            // exception on DatM.commit
            (!DatM.cancel; !Card.revert; !Log.print)
          )
        )
        +
        // exception on DatM.reserve
        (!DatM.cancel; !Card.revert; !Log.exEvent; !Log.print)
      )
      +
      // exception on Card.payment
      (!DatM.cancel; !Card.revert)
    )
    +
    // exception on Card.lookUp
    (!DatM.cancel; !Card.revert)
  )
  +
  // exception on DatM.preReserve
  (!Log.exEvent; !Log.print)
}
...
```

3.2 Analyzing the problem and sketching a solution

In this section, we discuss all the key aspects related to expressing/capturing exceptions and behavior protocols at the level of an ADL (Architecture Description Language). In our view, the driving facts are:

1. In ADLs, exceptions should be specified with a granularity of a method (most likely in the interface specifications).
2. In ADLs, the key abstractions the protocols are associated with are frame protocols.
3. Issuing a method call in a frame protocol means the call goes outside of the component.
4. Throwing an exception in a method means an abnormal end of the method call.
5. Because of (3) an exception has to be handled in the frame protocol of the component which issued the call, and, because of (4), such handling is a specific reaction of the calling component after receiving the exception. In principle, this reaction has to be reflected by an adequate "traffic" on the calling component's interfaces.

Obviously an abnormal end of a method `i2.m` call from interface `i` can be easily modeled by replacing the standard "end_of_call" response `!i.m`↓ by an exception response, e.g. `!i.e`↓. Moreover, the "abnormality" has to be reflected by abandoning the original protocol specifying the execution of m, i.e. the action `!i.e`↓ in the protocol `P` appearing in the context `?i2.m{P}` has to be the last action generated by `P`. However, as the example in Sect. 3.1 indicates, addressing these abnormalities by the standard behavior protocols means becomes cumbersome. Since any protocol can be interpreted as an abstraction of code, we can, for this purpose, advantageously adopt a Java inspired construct of the form `?i2.m{... throw !e`↓ `...}` with the meaning (informally put) `throw !e`↓ generates `!i.e`↓ and then the execution of `i2.m` internals directly jumps to the lexically nearest }. In a similar vein, for handling an exception in a caller's frame protocol ((5)), we can adopt a `try {P} catch {?i.e`↓`:Q}` construct with the meaning very similar to the interrupt operator in CSP (i.e. `P`$\triangle_i$`Q`): if the event at the beginning of `Q` occurs, then the execution of the process `P` is abandoned and the process `Q` executes further. Along these lines, the event `?i.e`↓ is the first one generated by the `catch {?i.e`↓`:Q}` construct.

However, we have to analyze exception throwing, propagating, and handling in all the (1)..(4) contexts below, since the methods are called across component boundaries and components can be nested. These four contexts represent all the situations on interface bindings related to a method call and an exception throwing and handling. These are client (1) and server (2) positions at a binding when no nesting is considered and the related situation when component nesting is taken into account. The latter are: nested server (3 - delegation) and nested client (4 - subsumption) positions at a binding.

(1) Client position (Figure 2)

Consider the component `X` which calls the method `a` on the interface `A`. If an exception `e` can be thrown by the call of `a` (i.e. thrown by `Y` in the setting of Fig. 2), the construct `try {... !A.a ...} catch {?A.e`↓`: ...}` is to be used in the frame protocol of `X` in order to handle the exception. An unhandled exception would cause an error (bad activity in terms of [1]).

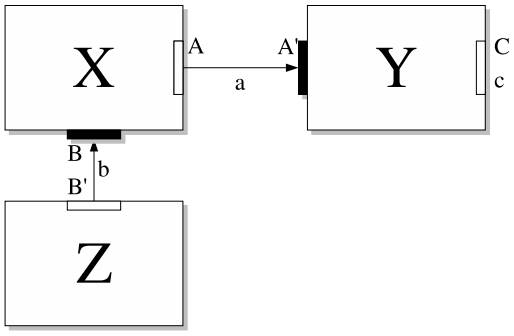

Fig. 2. Client-server

(2) Server position (Figure 2)

Consider the component `Y` accepting a call of `a` through the interface `A'`. In general, as mentioned above in this section, an exception in the execution of a is expressed by:

```
?A.a { ... throw !e↓; ... }
```

Based on the experience with our case studies, typical special cases of throwing an exception are:

1. An exception `e` is thrown due to an invalid actual parameter of `a` (invisible in protocols, but important for a credible abstraction). In protocols, this is typically expressed as
   ```
   ?A.a { null + throw !e↓; ... }
   ```
2. An exception is thrown due to a faulty return value in a nested call `!C.c`: (again invisible in protocols, but important for a credible abstraction). In protocols this is typically expressed as
   ```
   ?A.a { ... !C.c; (null + throw !e↓); ... }
   ```
   Since both in (1) and (2) the exception is a reaction on an "invisible" invalid value, it is a good practice to indicate the fact by choosing a mnemotechnical name for the exception (in Sect. 4, there are several examples of this method).
3. An exception is thrown in a `catch` construct. This is typical for exception propagation (even under a different name). For example, in
   ```
   ?A.a { ... try {!C.c} catch {?C.e1↓: throw !e2↓); ... }
   ```
   the `C.c` method can throw an `e1` exception, which is then converted into `e2`.

(3) Delegation (Figure 3)

Delegation basically means forwarding an acceptance of a call to an internal component[15]; in Fig. 3 the component `Y` delegates calls from the component `X` on the interface `A` to the interface `A'` in the internal component `ZA`. In principle, an exception `e` thrown in `ZA` in its method `a`, has to be delivered to the original caller, i.e. to the component `X`. Since the internals of `Y` are not visible to `X`, throwing of `e` should be specified not only in the frame protocol of `ZA` and but also of `Y`. Notice, however, that an exception thrown by the component `ZC` and handled by the component `ZA` would not be visible in the frame protocol of `Y`.

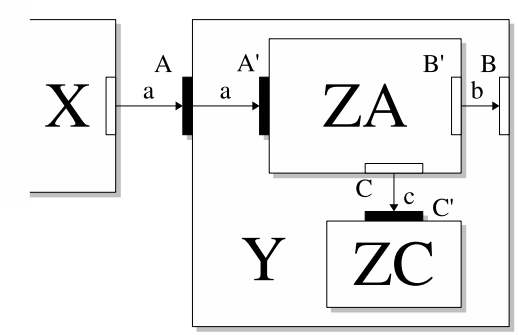

Fig. 3. Delegation

(4) Subsumption (Figure 4)

Subsumption basically means forwarding a call issued in an internal component to its parent component[15]; in Fig. 4, the component `ZA` subsumes the calls on the interface `A'` to the interface `A` in its parent component `X`.

Apparently, an exception thrown in `Y` is to be delivered to and handled by the caller, i.e. the component `ZA`. However, the component `X` is also in the client position with respect to `Y` (and, at a design stage, the internals of `X` do not have to be known). Therefore, handling of the exception has to be specified also in the frame protocol of `X`.

Since handling an exception in the frame protocol of `X` in general causes a "recovery communication" of `X` visible outside of `X`, potentially including a specific communication on its interface `B`. Obviously, this recovery communication should be adequately captured in the architecture protocol of `ZA` and `ZB` and, in particular, triggered by handling the exception in the frame protocol of `ZA`.

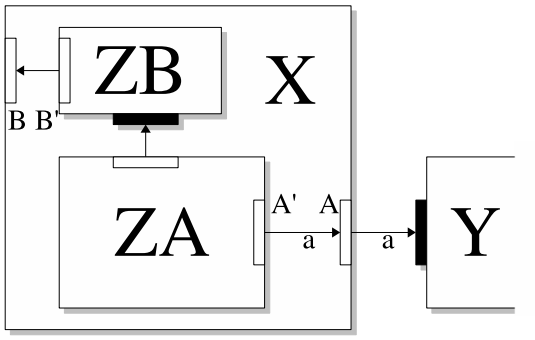

Fig. 4. Subsumption

3.3 Proposed solution - details

The main purpose of this section is to describe in more detail the semantics of the constructs introduced in Sect. 3.2 and to analyze the influence of these protocol enhancements on protocol compliance evaluation[1,3,15]. By convention, we will refer to exception handling based on these constructs as *Explicit try-catch* technique.

Throwing an exception

Syntax: `throw !exception_name↓`

This construct has to appear only in a protocol `P` written in the context of the form `i.a{P}`, i.e. inside the curly brackets abbreviation expressing call acceptance of a method `a`. In principle, `throw !exception_name↓` means that in the resulting trace the event `!i.a↓` modeling return from `a` is replaced by the event `!i.exception_name↓` and, at the same time, this is the last event generated by `P`. Should `P` contain nested method accepting constructs (such as `?i.b{Q}`), this principle is applied recurently.

For example, `?i.m{!a.x;?a.sl; throw !ex↓;!a.y}` would always generate the trace `?i.m↑,!a.x,?a.sl,!a.ex↓`. In a similar vein, `?i.m{X;(null + throw !ex↓); Y}` is equivalent to `?i.m↑; X;(!i.ex↓ + Y; !i.m↓)` for some protocols `X` and `Y`.

It should be emphasized that `!exception_name↓` is always the last event generated by `P`, even though `P` contains a `|` and/or `||` operator. For example, the traces generated by `?i.m{(!a.x;?a.sl; throw !ex↓;!a.y)||!a.z*}` include (the beginning resp. end of a trace is denoted by `<` resp. `>`):

```
<?i.m↑;!a.x↑;!a.x↓;?a.sl↑;!a.sl↓;!i.ex↓>
<?i.m↑;!a.z↑;!a.x↑;?a.sl;!a.z↓;!a.z;!i.ex↓>
<?i.m↑;!a.x↑;!a.z;!a.z;?a.sl;!a.z;!a.z;!a.z;!a.ex↓>
<?i.m↑;!a.x;!a.z↑;?a.sl↑;?a.z↓;!a.sl↓;!a.z↑;!a.ex↓>
```

Anyhow, the reason why `throw !exception_name↓` generates the last event in `P`, no matter how many parallel activities in `P` are specified, is that it is hard to define a "reasonable" semantics of more than one exception (the remaining parallel activities could also throw an exception). As an aside, by opting for "interrupting" all the parallel activities we basically follow the semantics chosen in CSP for the interrupt operator[10].

Catching an exception

Syntax:

`try` { A }
`catch` {?$\mathtt{i}_{1,1}$.`exception_name`$_{1,1}$↓, . . . , ?$\mathtt{i}_{1,m_1}$.`exception_name`$_{1,m_1}$↓ : B_1}
`catch` {?$\mathtt{i}_{2,1}$.`exception_name`$_{2,1}$↓, . . . , ?$\mathtt{i}_{2,m_2}$.`exception_name`$_{2,m_2}$↓ : B_2}
. . .
`catch` {?$\mathtt{i}_{n,1}$.`exception_name`$_{n,1}$↓,. . . , ?$\mathtt{i}_{n,m_n}$.`exception_name`$_{n,m_n}$↓: B_n}

where A, B_j are protocols and $\mathtt{i}_{ij}$ are interfaces. If a `throw !exceptionname`$_{ij}$↓ is applied in A in a context

`try { ... !A.a ...} catch {?`$\mathtt{i}_{ij}$`.exception_name`$_j$`↓: ...:`B_i`}`,

then the next event generated by the `try` construct is the first event specified by B_i. For simplicity, all the exceptions which could be thrown in the `try` construct have to be listed exactly once in one of the `catch` parts of the construct.

Influence on compliance evaluation. The exception-related constructs preserve the semantics of the operators defined for behavior protocols (in particular the semantics of the composition and consent operators important for composition error detection and compliance evaluation[1,15]).

Unhandled (uncaught) exceptions are captured statically by the protocol checker (information about the possible exceptions have to be a part of interface specification in ADL). An improper/non-existent reaction to an exceptions is typically captured as a bad activity error.

It can be easily shown that these constructs are without difficulty captured by the LTS representing a behavior protocol. For example, the LTS representing

```
try { !i.a1; !i.a2; !i.a3}
catch { ?i.e1↓: !i.b}
catch { ?i.e2↓: !i.c}
```

can be easily constructed by adding transitions to the states representing the methods' calls in which `e1` and `e2` can be returned: these transitions will lead to the LTS representation of the `e1` and `e2` handlers (Fig. 5). Since the resulting LTS remains a finite automaton, the finite trace-based semantics of the behavior protocol operators is preserved.

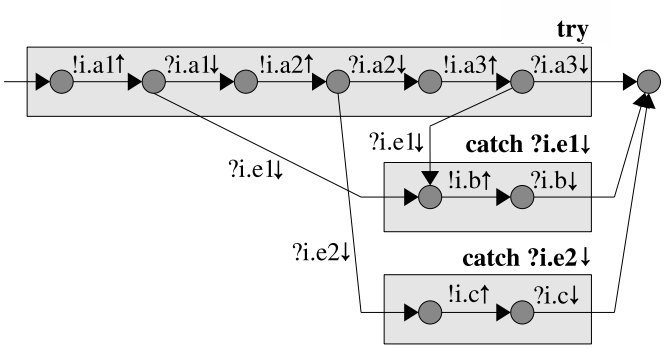

Fig. 5. Transition diagram

4 Case study

In this section, using again the example from Fig. 1, we illustrate how the exception-related constructs simplify specification of exceptions in the behavior protocols of the components introduced in Sect. 2.

The frame protocol of the Ticket manager is shown below. The component was already described in the Sect. 2; here we present a slightly more detailed version which includes the initialization of other components and the check of the `lookUp` method return value.

Ticket manager frame protocol

```
?Usr.init { !Card.init; !DatM.init};
(
  ?Usr.buyTicket {
    try {
      !DatM.preReserve; !Card.lookUp;
      null + (!Card.payment; !DatM.reserve; !DatM.commit)
    }
    catch { ?DatM.PreReservationException↓:
      !Log.exEvent;}
    catch { ?Card.NetworkException↓:
      !DatM.cancel; !Card.revert;
      throw !NetworkException↓}
    catch { ?DatM.DatabaseException↓, ?DatM.ReservationException↓:
        !DatM.cancel; !Card.revert; !Log.exEvent };
    !Log.print;
  }
  +
  ?Usr.returnTicket {
    try { !Card.revert; !DatM.free; !DatM.commit }
    catch {?DatM.DatabaseException↓:
      !DatM.cancel; !Log.print}
  }
)*;
?Usr.finish {!DatM.finish; !Card.finish}
```

Below is the frame protocol of the VISA component. After being initialized by accepting an `init` call, the "business" stage takes place: `lookUp`, `payment`, and `revert`. The `lookUp` and `payment` methods may throw an exception due to the problem on the network (`null + throw !NetworkException↓`). The `lookUp` method also verifies the card number via the verify method on the Blacklist interface. If the verification yields a negative result, `ListException` is thrown and validity is re-checked by the operator (call of `askValidity` on the Operator interface).

VISA frame protocol

```
?Card.init;
(
  ?Card.lookUp {
    try { !Blacklist.verify }
    catch { ?BlackList.ListException↓: !Oper.askValidity};
    null + throw !NetworkException↓
  }
  +
  ?Card.payment {
    null + throw !NetworkException↓}
  +
  ?Card.revert
  )*;
?Card.finish
```

In a similar vein, the frame protocol of Database manager indicates that the `preReserve`, `reserve` and `commit` methods can be alternatively called after initialization. All of them communicate with the Storage component via a `!Strg.Access` call which can return `StorageException`. Notice that this exception is converted to the `PreReservationException` resp. `ReservationException` consequently delivered to the caller of `preReserve` resp. of `reserve` or `commit`.

Database manager frame protocol

```
?DatM.init { !Strg.init };
(
  ?DatM.preReserve {
    try { !Strg.Access* }
    catch { ?Strg.StorageException↓:
      throw !PreReservationExcpetion↓}
  }
  +
  ?DatM.reserve {
    try { !Strg.Access* }
    catch { ?Strg.StorageException↓:
      throw !ReservationException↓}
  }
  +
  ?DatM.commit {
    try { !Strg.Access* }
    catch { ?Strg.StorageException↓:
      throw !ReservationException↓}
  }
  +
  ?DatM.cancel
)*;
?DatM.finish { !Strg.finish }
```

Storage frame protocol

```
?Strg.init;
(
  ?Strg.access { null + throw !StorageException↓}
)*;
?DatM.finish
```

The frame protocol of Reservation describes the communication with the environment of the whole reservation application. Notice that the exceptions which are thrown and handled inside the component are naturally not visible at this level, but `NetworkException` is propagated through the Reserv interface so that it has to appear in the frame protocol in the `throw` construct. On the other hand, `ListException` is handled in this frame protocol as `null}` since its handling does not require external component communication (as an aside, details of its handling are visible the VISA frame protocol - `!BlackList.test` is subsumed from VISA). In contrast, if a OperatorVerification component were outside Reservation (Fig. 6), details of `ListException` handling would be visible in the Reservation frame protocol, as illustrated in it by the comment line.

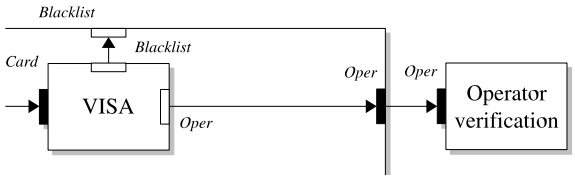

Fig. 6. Modified reservation component

Reservation frame protocol

```
?Reserv.init;
(
  ?Reserv.buyTicket {
    !Log.exEvent;
    !Log.print
    +
    (
      try { !BlackList.test }
      catch { ?BlackList.ListException↓: null};
                // catch { ?BlackList.ListException↓: !Oper.askValidity}
      (!Log.exEvent + null);
      !Log.print + throw !NetworkException↓
    )
  }
  +
  ?Reserv.returnTicket
)*;
?Reserv.finish
```

5 Evaluation

This work was inspired by our experience gained during our attempt to a apply behavior protocols to a non-trivial, real-life component-based application. We had chosen the Speedo project[22] available from the ObjectWeb consortium as an open source implementation of the Sun JDO specification[23]. The implementation is based on the FRACTAL component model[7] and is heavily using the Perseus persistence framework[14]. Together, behavior protocols of 26 components were written. Our experiences has been that without an explicit notation for exception handling, protocols are very hard to read and comprehend, and furthermore, the correspondence between the behavior specification and code is very hard to trace. We support this claim be the figures provided in Fig. 7. Here, the length of behavior protocol specification is given for four specific techniques of expressing exceptions via behavior protocols. The "Ignoring exceptions" techniques means specifying behavior in a way which does not consider exceptions at all. The "Explicit try-catch" technique is based on the behavior protocol extensions described in Sect. 3.2, while "Intrinsic exceptions" and "Approximation by alternative" are the methods described in Sect. 3.1.

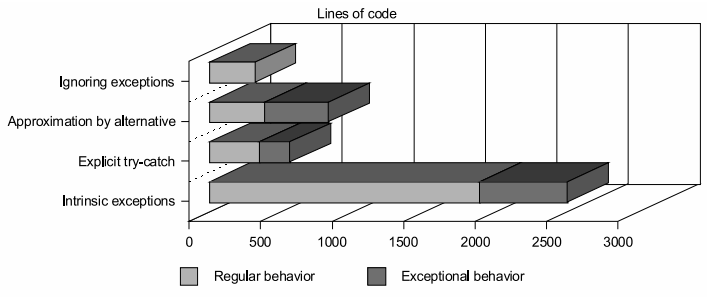

Fig. 7. Description complexity of the Speedo project formalism

Each bar of the graph is divided into two parts to indicate the number of lines specifying the "regular" behavior (gray) and exception-related behavior (black). From the chart it is clearly visible how significantly the proposed "Explicit try−catch" construct shortens behavior specification. Both "Approximation by alternative" and "Explicit try−catch" do not cause any significant grows of the "regular" part of the behavior specification, in contrast to the "Intrinsic exceptions" technique where often some of the specification sections have to be repeated. Notice also that "Approximation by alternative" causes grows of the exception-related behavior specification in comparison with "Explicit try−catch".

6 Related work

There are many publications on exception handling, however not many of them are related to exceptions at a level of abstraction higher than source code.

In [20] the authors employ the C2 architectural style featuring composition contracts. Components have top and bottom interfaces connected via connectors responsible for routing and filtering asynchronous messages. There are two types of messages - a request message and a notification message depending on whether the message flows up or down though the system. This is very similar to our request-response notation. The composite contract (a service-implementing component) ends either with a normal notification or an exceptional notification. In the latter case, an exception handler component is activated. If the exception recovery is successful, an abort notification is generated; otherwise a failure notification is generated and the component may be left in an inconsistent state. In this approach, the contract component "remotely" corresponds to our `try` construct and exception component to the `catch` construct, however, the philosophy of component hierarchy is different compared to ours and there is no behavior specification at the level of the whole component.

The static source analyzing tool PREfast[13] checks all the execution traces for possible erroneous behavior (typically null reference, memory leaks). In the context of this paper, it is interesting that during exception propagation some (predefined) functional failures are detected, such as missing memory deallocation and resource unlocking. This property is checked by our approach implicitly - communication errors would be detected in the behavior composition process[3].

Session types are used for describing behavior of CORBA IDL in [24]. The approach of behavior description is similar to our interface protocol (protocols restricted to an

interface), with a different syntax though. An exception is expressed in the specification of potential responses of a method. However, if the method can raise more exceptions, the same label is used for each of them.

A CSP based exception handling is introduced in [9]. The exception operator ($\overrightarrow{\triangle}$), is inspired by the CSP interrupt operator $\triangle_i$[10]. While $\mathtt{P}\triangle_i\mathtt{Q}$ means preemption of `P` on an externally coming event `i` and continuation by `Q` (`i` is the first event of `Q`), the exception operator considers in $\mathtt{P}\overrightarrow{\triangle}\mathtt{Q}$ the event `i` as an internal event and therefore `Q` can be interpreted as an exception handler and `P` as a `try` construct. Since in our proposed extension of behavior protocols the composition of two components' behavior also yields an internal action $\tau\mathtt{e}$ (one of the components throws an exception $\mathtt{!e}\downarrow$ and another one accepts it via $\mathtt{?e}\downarrow$ in a `catch` construct), the approaches are similar in this respect. However, there are significant differences. In CSP, interrupts can occur without an intervention of the original process `P`, thus being similar to hardware interrupts. In our approach, an exception is triggered by invoking a method call and it has to be an expected event. Additionally, one catch block can handle more that one exception to avoid repeating of the same handling routine if an identical reaction is desirable. Also exception handling can be subject of compliance tests of both horizontal and vertical contracts (Sect. 2).

7 Conclusion

The key contributions of this paper include:

(i) An analysis of the role and importance of exceptions in behavior specification of software components is given and it is shown how behavior protocols can be extended to handle exceptions in an efficient way in terms of readability, comprehension, and the size of a behavior specification.

(ii) This claim is supported by providing experimental results from a real-life case study of applying different exception handling techniques based on behavior protocols. From these experiments, it is clearly visible how significantly the proposed behavior protocol extension by an explicit exception handling construct shortens the behavior specification of a non-trivial component-based application.

Acknowledgments

We would like to give a special credit to our colleagues Jiri Adamek and Vladimir Mencl for their valuable comments, and to Jan Kofron and Pavel Jezek for another non-trivial case study and a number of suggestions.

References

1. J. Adamek, F. Plasil: Component Composition Errors and Update Atomicity: Static Analysis, Journal of Software Maintenance and Evolution: Research and Practice 17(5), pp. 363-377, 2005 2, 7, 10, 11
2. J. Adamek, F. Plasil: Erroneous Architecture is a Relative Concept, in Proceedings of SEA conference, Cambridge, MA, USA, ACTA Press 2004 2, 3
3. J. Adamek, F. Plasil: Partial Bindings of Components − any Harm?, in the Proceedings of APSEC 2004, IEEE Computer Society, pp. 632−639, Nov 2004 2, 10, 15

4. L. de Alfaro, T. A. Henzinger: Interface Automata, in Proceedings of the 9^th Annual ACM Symposium on Foundations of Software Engineering (FSE), 2001 1
5. R.J. Allen, D. Garlan: A Formal Basics For Architectural Connection, ACM Transactions on Software Engineering and Methology, Jul 1997 1
6. W. DePrince jr., C. Hofmeister: Enforcing a lips Usage Policy for CORBA Components, in proceedings of EUROMICRO'03, Sep 2003 1
7. Fractal component model: http://fractal.objectweb.org/ 14
8. D. Harel: Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming 8, Elsevier Science Publishers B.V., 1987 1
9. G.H. Hilderink: Managing Complexity of Control Software through Concurrency, PhD thesis, University of Twente, The Netherlands, ISBN 90-365-2204-8, May 2005 16
10. C.A.R. Hoare: Communicating Sequential Processes, Prentice-Hall International, UK, Ltd., ISBN 0-13-153271-5, 1985 10, 16
11. J. Magee, J. Kramer: Concurrency: State models & Java programs, John Wiley & Sons Ltd, ISBN 0-471-98710-7, 1999 1
12. Object Management Group: UML 2.0 Infrastructure Final Adopted Specification, OMG document ptc/03-09-15, Sep 2003 1
13. PREfast: http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx 15
14. Perseus persistence framework: http://perseus.objectweb.org/ 14
15. F. Plasil: Enhancing Component Specification by Behavior Description − the SOFA Experience, in Proceedings of the 4th WISICT 2005, A volume in the ACM, Computer Science Press, Trinity College Dublin, Ireland, pp. 185−190, Jan 2005 8, 9, 10, 11
16. F. Plasil, S. Visnovsky: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002 1, 2, 3, 4
17. F. Plasil, S. Visnovsky, M. Besta: Bounding Behavior via Protocols, in Proceedings of TOOLS USA '99, 1999 3
18. R.H. Reussner, S. Becker, V. Firus: Component Composition with Parametric Contracts, Tagungsband der Net.ObjectDays, 2004 1
19. H.W. Schmidt, B. J. Kramer, I. Poernomo, R. Reussner: Predictable Component Architectures Using Dependent Finite State Machines, Proceedings of the 9th International Workshop in Radical Innovations of Software and Systems Engineering in the Future, LNCS Springer-Verlag, ISBN 3-540-21179-9, Vol 2941, 2004 1
20. R.M. Silva, P.A.C. Guerra, C.M.F. Rubira: Component Integration using Compositions Contracts with Exception Handling, in Proceedings of ECOOP2003 Workshop on Exception Handling in Object-Oriented Systems, TR 03-028 Univ. of Minnesota, Dept. of Comp.Sci., Jul 2003 15
21. SOFA project: http://sofa.objectweb.org/, http://nenya.ms.mff.cuni.cz/projects.phtml?p=sofa\&q=0 3
22. Speedo: http://speedo.objectweb.org/ 14
23. Sun JDO specification: http://java.sun.com/products/jdo/ 14
24. A. Vallecillo, V.T. Vasconcelos, A. Ravara: Typing the Behavior of Objects and Components using Session Types, 1st International Workshop on Foundations of Coordination Languages and Software Architectures. Electronic Notes in Theoretical Computer Science, 2002 15
25. K. Wallnau: Volume III: A Technology for Predictable Assembly from Certifiable Components, Technical Report CMU/SEI-2003-TR-009, Apr 2003 1