Modeling of Component Environment in Presence of Callbacks and Autonomous Activities

Pavel Parizek[1] and Frantisek Plasil[1,2]

[1] Charles University in Prague, Faculty of Mathematics and Physics,
Department of Software Engineering, Distributed Systems Research Group
{parizek,plasil}@dsrg.mff.cuni.cz
http://dsrg.mff.cuni.cz
[2] Academy of Sciences of the Czech Republic,
Institute of Computer Science

Abstract. A popular approach to compositional verification of component-based applications is based on the assume-guarantee paradigm, where an assumption models behavior of an environment for each component. Real-life component applications often involve complex interaction patterns like callbacks and autonomous activities, which have to be considered by the model of environment's behavior. In general, such patterns can be properly modeled only by a formalism that (i) supports independent atomic events for method invocation and return from a method and (ii) allows to specify explicit interleaving of events on component's provided and required interfaces - the formalism of behavior protocols satisfies these requirements. This paper attempts to answer the question whether the model involving only events on provided interfaces (calling protocol) could be valid under certain constraints on component behavior. The key contribution are the constraints on interleaving of events related to callbacks and autonomous activities, which are expressed via syntactical patterns, and evaluation of the proposed constraints on real-life component applications.

Key words: assume-guarantee reasoning, behavior protocols, modeling of environment behavior, callbacks, autonomous activities

1 Introduction

Modern software systems are often developed via composition of independent components with well-defined interfaces and (formal) behavior specification of some sort. When reliability of a software system built from components is a critical issue, formal verification such as program model checking becomes a necessity. Since model checking of the whole complex ("real-life") system at a time is prone to state explosion, compositional methods have to be used. A basic idea of compositional model checking [6] is the checking of (local) properties of isolated components and inferring (global) properties of the whole system from the local properties. This way, state explosion is partially addressed, since a single isolated component typically triggers a smaller state space compared to the whole system.

A popular approach to compositional model checking of component applications is based on the assume-guarantee paradigm [18]: For each component subject to checking, an assumption is stated on the behavior of the component's environment (e.g. the rest of a particular component application); similarly, the "guarantee" are the properties to hold if the component works properly in the assumed environment (e.g. absence of concurrency errors and compliance with behavior specification). Thus, a successful model checking of the component against the properties under the specific assumption guarantees the component to work properly when put into an environment modeled by the assumption.

Specific to program model checkers such as Java PathFinder (JPF) [21] is that they check only complete programs (featuring `main()`). Thus checking of an isolated component (its implementation, i.e. for instance of its Java code) is not directly possible ([17], [10]), since also its environment has to be provided in the form of a program (code). Thus, program model checking of a primitive component is associated with the *problem of missing environment* [14]. A typical solution to it in case of JPF is to construct an "artificial" environment (Java code) from an assumption formed as a behavior model as in [14][20], where the behavior model is based on LTS defined either directly [10], or in the formalism of behavior protocols [19]. Then, JPF is applied to the complete program composed of the component and environment.

In general, real-life component applications feature circular dependencies among components involving complex interaction schemes. Nevertheless, for the purpose of program model checking of an isolated component, these schemes have to be abstracted down to interaction patterns between the component and its environment pairs. Based on non-trivial case studies [1][8], we have identified the following four patterns of interaction between a component C and its environment E to be the most typical ones (*C-E patterns*):

a) *synchronous callback* (Fig. 1a), executed in the same thread as the call that triggered the callback;

b) *asynchronous callback* (Fig. 1b), executed in a different thread than the trigger call;

c) *autonomous activity* (Fig. 1c) on a required interface, which is performed by an inner thread of the component;

d) *synchronous reaction* (Fig. 1d) to a call on a component's provided interface.

In Fig. 1, each of the sequence diagrams contains activation boxes representing threads (`T1` and `T2`) running in the component and environment in a particular moment of time. More specifically, in Fig. 1a, `m` denotes a method called on the component by the environment, and `t` denotes the trigger (invoked in `m`) of the callback `b`; note that all calls are performed by the same thread (`T1`). As to Fig. 1b, the only difference is that the callback `b` is asynchronous, i.e. it is performed by a different thread (`T2`) than the trigger `t`. In case of Fig. 1c, the

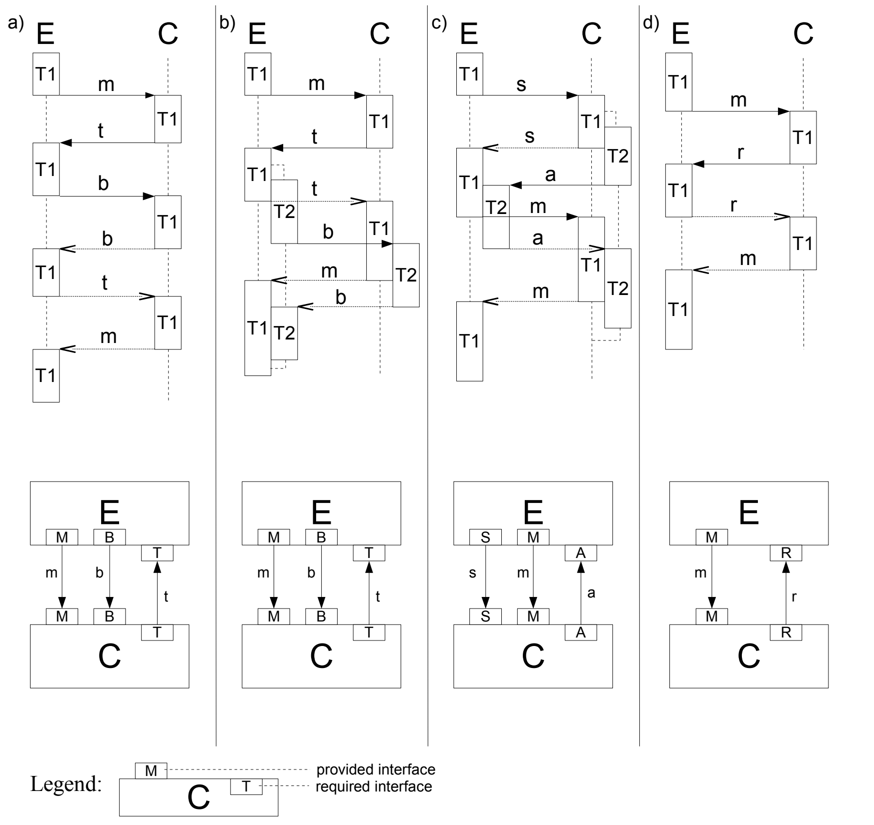

Fig. 1. Interaction patterns (C-E patterns) between a component and its environment

method `s` called on the component by the environment (in thread `T1`) starts an autonomous activity performed by an inner thread (`T2`), which calls the method `a` of the environment. The latter overlaps with the call of `m` issued by the environment. Finally, in Fig. 1d, `r` denotes a synchronous reaction to the call of the method `m` issued by the environment (both performed in the same thread).

These sequence diagrams clearly show that proper modeling of these C-E patterns via a specific formalism is possible only if the formalism allows to explicitly model the interleaving of method invocations and returns from methods on *C*'s provided and required interfaces. Specifically, a method call as whole cannot be modeled as an atomic event (like in [10]); instead, independent constructs for method invocation (*invocation event*), return from a method (*return event*) and method execution (*method body*) have to be supported by the formalism. We say that a model of environment's behavior is *valid* if it precisely describes all occurrences of the C-E patterns in the interaction between a component and its environment.

The formalism of behavior protocols [19], developed in our group, supports independent invocation and return events on *C*'s provided and required interfaces (details in Sect. 2) and therefore allows to model all the C-E patterns properly. In our former work, we introduced two specific approaches to modeling of environment's behavior: *inverted frame protocol* [14] and *context protocol* [15]. Both of them are generic, i.e. not limited to any particular communication pattern between E and C, and valid. The key difference between these two is that the inverted frame protocol models E that exercises C in all ways it was designed for (*maximal-calling environment*), while the context protocol models the actual use of C in the given context of a component-based application (*context environment*). Specifically, the context protocol may be simpler than the inverted frame protocol, e.g. in terms of level of parallelism (i.e. the assumption on environment behavior is weaker), if the particular application uses only a subset of C's functionality.

Unfortunately, the actual JPF model checking of a component combined with an environment determined by any of these two modeling approaches is prone to state explosion, in particular for two reasons:

(1) Java code of E is complex, since it has to ensure proper interleaving of the events on C's provided interfaces triggered by E with the events on C's required interfaces triggered by C itself. Technically, since there is no direct language support for expressing acceptance of a method call depending upon calling history in Java, the interleaving has to be enforced indirectly, e.g. via synchronization tools (`wait`, `notify`) and state variables.

(2) As to the context environment, its construction is also prone to state explosion, since the context protocol is derived from behavior specifications of the other components in a particular component application via an algorithm similar to the one employed in behavior compliance model checking [11]. In [15] we presented a syntactical algorithm for derivation of a context protocol, which has a low time and space complexity; however, it does not support autonomous activities and does not handle cycles in architecture (and thus callbacks) properly in general.

The issues (1) and (2) are particularly pressuring when C is designed to handle a high level of parallel activities (threads). Then, this has to be reflected in E to exercise C accordingly. To alleviate the state explosion problem associated with these issues we proposed in [16] a simplified approach to modeling environment behavior: *calling protocol*. Roughly speaking, a calling protocol models precisely only the events on C's provided interfaces, i.e. it models only the calls issued by E, under the assumption that the calls issued by C are accepted by E at any time (and in parallel) - this is an overapproximation of the desired behavior of E. Thus the Java code of E is simple, since it does not have to ensure proper interleaving of the events on C's provided and required interfaces. On the other hand, capturing this interleaving is necessary for an appropriate modeling of the C-E patterns in general, and thus for validity of a calling protocol-based model of environment's behavior. An open question is whether there are constraints on behavior of C under which the calling protocol-based approach could provide a valid model of E.

1.1 Goals and Structure of the Paper

The goal of this paper is to answer the question whether the calling protocol-based approach can provide a valid model of environment behavior in the context of the C-E patterns, if, in the component's behavior specification, certain constraints are imposed on the sequencing and interleaving of the C-E events with other events on the component interfaces.

The structure of the paper is as follows. Sect. 2 provides an overview of the formalism of behavior protocols and its use for modeling of environment behavior. Sect. 3 presents the key contribution of the paper - an answer to the question of validity of the calling protocol-based approach under certain constraints on component behavior and an algorithm for automated construction of a valid calling protocol-based model of environment's behavior. Sect. 4 shows experimental results and the rest contains evaluation, related work and a conclusion.

2 Behavior Protocols

The formalism of behavior protocols - a simple process algebra - was introduced in [19] as a means of modeling behavior of software components in terms of traces of atomic events on the components' external interfaces. Specifically, a *frame protocol* FP_C of a component C is an expression that defines C's behavior as a set $L(FP_C)$ of finite traces of atomic events on its provided and required interfaces.

Syntactically, a behavior protocol reminds a regular expression over an alphabet of atomic events of the form `<prefix><interface>.<method><suffix>`. Here, the prefix `?` denotes acceptance, while `!` denotes emit; likewise, the suffix ↑ denotes a method invocation and ↓ denotes a return from a method. Thus, four types of atomic events are supported: `!i.m`↑ denotes emitting of a call to method `m` on interface `i`, `?i.m`↑ acceptance of the call, `!i.m`↓ emitting of return from the method, and, finally, `?i.m`↓ denotes acceptance of the return. Several useful shortcuts are also defined: `!i.m{P}` stands for `!i.m`↑ `; P ; ?i.m`↓ (*method call*), and `?i.m{P}` stands for `?i.m`↑ `; P ; !i.m`↑ (*method acceptance*). Both in `!i.m{P}` and `?i.m{P}`, a protocol `P` models a *method body* (possibly empty). As for operators, behavior protocols support the standard regular expression operators (sequence (`;`), alternative (`+`), and repetition (`*`)); moreover, there are two operators for parallel composition: (1) Operator `|`, which generates all the interleavings of the event traces defined by its operands; the events do not communicate, nor synchronize. It is used to express parallel activities in the frame protocol of C. (2) Operator ∇_S ("consent"), producing also all interleavings of the event traces defined by its operands, where, however, the neighboring events from S (with "opposite" prefix) are complementary - they synchronize and are forced to communicate (producing internal action τ similar to CCS and CSP). An example of such complementary events would be the pair `!I.m`↑ and `?I.m`↑. This operator is used to produce the composed behavior of cooperating components, while S comprises all the events on the component's bindings. Moreover, it also indicates communication errors (deadlock and "bad activity" - there is no complementary event to `!I.m`$\uparrow$ in a trace, i.e. a call cannot be answered).

a) synchronous callback:	FP_{Ca} = `?M.m {!T.t{?B.b}}`
b) asynchronous callback:	FP_{Cb} = `?M.m`$\uparrow$`;!T.t`$\uparrow$`;?T.t`$\downarrow$`;?B.b`$\uparrow$`;!M.m`$\downarrow$`;!B.b`$\downarrow$
c) autonomous activity:	FP_{Cc} = `?S.s;!A.a`$\uparrow$`;?M.m`$\uparrow$`;?A.a`$\downarrow$`;!M.m`$\downarrow$
d) synchronous reaction:	FP_{Cd} = `?M.m {!R.r}`

Table 1. Frame protocols of C in Fig. 1

Using behavior protocols, a quite complex behavior can be modeled - see, e.g., [1] for a behavior model of a real-life component application. Advantageously, it is possible to model the explicit interleaving of events on both the provided and required interfaces of a component in its frame protocol. Specifically, the frame protocol of C in Fig. 1 in the alternatives a) - d) would take the form as in Tab. 1.

2.1 Modeling Environment via Behavior Protocols

Consider again the missing environment problem and the setting on Fig. 1. Obviously, the environment of an isolated component C can be considered as another component E bound to C. Thus the model of E's behavior can be a frame protocol of E. Since a required interface is always bound to a matching provided interface, the former issuing calls and the latter accepting calls, the corresponding events in both frame protocols ought to be complementary. For example, the frame protocols of E in Fig. 1 in alternatives a) - d) would have the form as in Tab. 2.

a)	FP_{Ea} = `!M.m {?T.t{!B.b}}`
b)	FP_{Eb} = `!M.m`$\uparrow$`;?T.t`$\uparrow$`;!T.t`$\downarrow$`;!B.b`$\uparrow$`;?M.m`$\downarrow$`;?B.b`$\downarrow$
c)	FP_{Ec} = `!S.s;?A.a`$\uparrow$`;!M.m`$\uparrow$`;!A.a`$\downarrow$`;?M.m`$\downarrow$
d)	FP_{Ed} = `!M.m {?R.r}`

Table 2. Frame protocols of E in Fig. 1

Obviously, an event issued by E (such as `!M.m`$\uparrow$) has to be accepted by C (such as `?M.m`$\uparrow$) at the right moment and vice versa. As an aside, this (behavior compliance [19]) can be formally verified by parallel composition via consent, $FP_E \nabla_S FP_C$, which should not indicate any communication error; for $FP_{Eb} \nabla_S FP_{Cb}$ this is obviously true, since FP_{Eb} was created by simply replacing all `?` by `!` and vice versa - FP_{Eb} is the *inverted frame protocol* (FP_{Cb}^{-1}) of C_b. Because of that and since here S comprises all events on the interfaces `M`, `T` and `B`, the consent operator produces traces composed of τ only.

In general, any protocol FP_E for which $FP_E \nabla_S FP_C$ does not yield any composition error is called *environment protocol* of C, further denoted as EP_C. In

Sect. 1, we proposed three specific techniques to construct C's environment protocol: (i) inverted frame protocol (EP_C^{inv}), (ii) context protocol (EP_C^{ctx}) and (iii) calling protocol (EP_C^{call}). Event though these techniques aim at "decent" exercising of C, an environment protocol may be very simple, designed to help check a specific property. Assume for instance that the interface `M` of C in Fig. 1 features also a method `x` and FP'_{Ca} = `?M.m {!T.t{?B.b}} + ?M.x`. Then, EP'_{Ca} = `!M.x` would be an environment protocol since $EP'_{Ca} \nabla_S FP'_{Ca}$ does not yield any composition error.

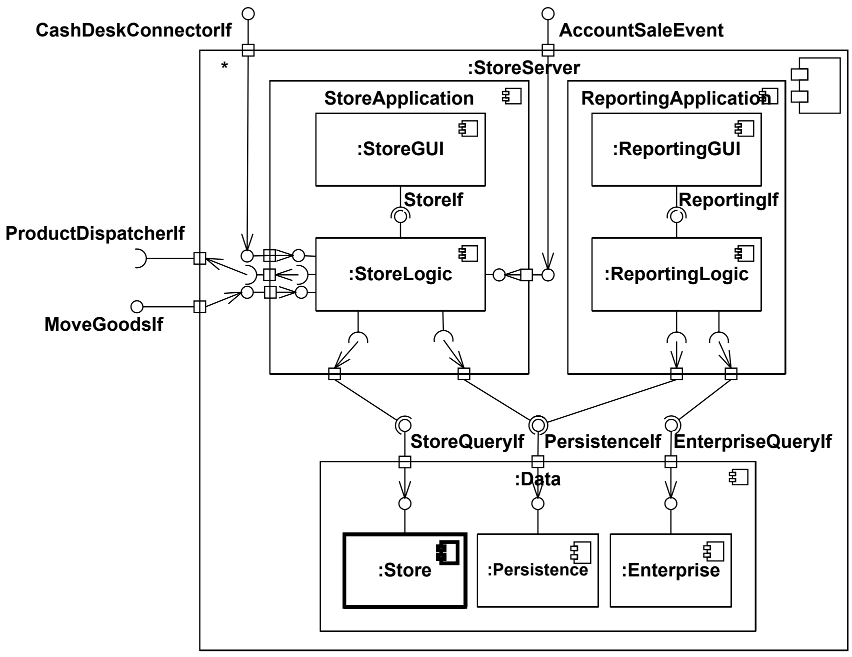

Fig. 2. Architecture of the `StoreServer` component

Nevertheless, the three techniques (i) - (iii) are much more of practical importance; below, they are illustrated on a part of the component architecture created in our group for the solution to the CoCoME assignment [8] (Fig. 2); the solution was based on the Fractal component model [3] and behavior protocols. Here we focus especially on the `Store` component, the functionality of which is not fully used by `StoreApplication` that accesses `Store` indirectly via `Data`. Specifically, the actual use of `Store` employs only a subset of the traces allowed by its frame protocol FP_{Store} (Fig. 3a); for example, FP_{Store} states that it is possible to call any method of the `StoreQueryIf` interface at most four times in parallel; however, assume that the `queryProductById` and `queryStockItem` methods on this interface are (indirectly) called three times in parallel by `StoreApplication` and the other methods are called only twice in parallel, or not at all in parallel. Therefore, the context protocol EP^{ctx}_{Store} (Fig. 3c) of `Store` is much simpler in terms of level of parallelism than its inverted frame protocol EP^{inv}_{Store} (Fig. 3b). Since the `Store` component has no required interface, its calling protocol EP^{call}_{Store} is equal to its context protocol, thus being obviously a valid model of the `Store`'s environment behavior in this special case.

a) FP_{Store} = (
```
    ?StoreQueryIf.queryProductById +
    ?StoreQueryIf.queryStockItem +
    # calls of other methods on StoreQueryIf follow
  )*
  |
  # the fragment above repeated three more times
```

b) EP^{inv}_{Store} = (
```
    !StoreQueryIf.queryProductById +
    !StoreQueryIf.queryStockItem +
    # calls of other methods on StoreQueryIf follow
  )*
  |
  # the fragment above repeated three more times
```

c) EP^{ctx}_{Store} = (!StoreQueryIf.queryStockItem* ; ...)*
```
  |
  (
    !StoreQueryIf.queryProductById*
    +
    !StoreQueryIf.queryStockItem*
  )*
  |
  !StoreQueryIf.queryProductById*
  |
  (
    ... ;
    (
      ( !StoreQueryIf.queryProductById*; ... )
      +
      ( ... ; !StoreQueryIf.queryStockItem* )
      +
      ...
    )
  )*
```

Fig. 3. a) a fragment of the frame protocol of `Store`; b) a fragment of the inverted frame protocol of `Store`; c) a fragment of the context protocol of `Store`

In summary, the basic idea of the techniques (i)-(iii) for construction of a model of C's environment (E) behavior is as follows:

Re (i) The inverted frame protocol EP_C^{inv} of a component C is constructed directly from the component's frame protocol FP_C by replacing all the prefixes `?` by `!` and vice versa.

Re (ii) The component's context protocol EP_C^{ctx} is derived via consent composition of the frame protocols of all the other components bound to C at the same level of nesting and the context protocol (or inverted frame protocol) of the C's parent component (if there is one).

Re (iii) The calling protocol can be derived in two ways: either via syntactical omitting of events on required interfaces from the inverted frame protocol or context protocol, or directly from the frame protocols of all the components in an architecture (including the one subject to checking) via a syntactical algorithm described in Sect. 3.2. In both cases, events on C's required interfaces, i.e. calls of E from C, are modeled implicitly in such a way that they are allowed to happen at any time and in parallel with any other event on any C's interface; technically, the environment protocol based on a calling protocol takes the form

EP_C^{call} = `<calling protocol> | ?m1* | ... | ?m1* |` (E1)
`| ?m2* | ... | ?mN*`

where `m1`, ..., `mN` represent methods of the component's required interfaces (obviously several instances of the same method can be accepted in parallel; nevertheless, the number `N` and the number of appearances of each `?mi*` have to be finite). Such EP_C^{call} is compliant with FP_C (assuming that compliance holds for frame protocols of all components in the application, which C belongs to), i.e. there are no communication errors, for the following reasons: (a) an environment modeled by EP_C^{call} calls C only in a way allowed by FP_C, since EP_C^{call} is derived from the frame protocols of the components cooperating with C at the same level of nesting (assuming their behavior is compliant); (b) an environment modeled by EP_C^{call} can accept any call from C at any time and in parallel with any other event on a C's interface (both provided and required).

3 Calling Protocol vs. Callbacks and Autonomous Activities

As indicated at the end of Sect. 1, an environment protocol based on a calling protocol (EP_C^{call}) is an imprecise model (overapproximation) of E's behavior in general, since it models in detail only the events on the C's provided interfaces and assumes a generic acceptance of calls on the required interfaces. Therefore, specifically, it is not possible to model detailed interleaving of events on these interfaces, which is necessary for proper modeling of callbacks and autonomous activities.

In this section, we propose certain syntactical constraints on C's frame protocol FP_C to ensure that no other events than those related to synchronous reactions, triggers of callbacks, and autonomous activities take place on the required interfaces of C; also, we answer the question whether EP_C^{call} can be a valid model of E's behavior if these constraints are satisfied.

The key idea is to express the constraints on FP_C via the following syntactical schemes (for simplicity, names of interfaces are omitted in event identifications):

(A) To express synchronous callbacks (and synchronous reactions) correctly, the constraint is that in FP_C the events corresponding to a particular callback `b` and a trigger `t` for `b` have to be nested according to the scheme

$FP_C = \alpha_1\ op_1$ `?m{` $\alpha_2\ op_2$ `!t{?b}` $op_3\ \alpha_3$`}` $op_4\ \alpha_4$ (A1)

where α_i may involve only synchronous reactions (C-E pattern (d)) and arbitrary behavior protocol operators except consent (∇), and op_i is either the sequence operator (`;`) or the alternative operator (`+`). Specifically, the frame protocol $FP'_C = \alpha_1 op_1$ `?m ;` α_2 `; !t ;` α_3 `; ?b` $op_2 \alpha_4$, which would be the only option when using the LTS-based approach of [10], violates the constraint. An example of a frame protocol that satisfies the constraint is

FP'_{Ca} = `?m1{!r1} ; ?m2{!t{?b} + !r2} ; ?m3{!r3}` (EX-A1)

(B) To express asynchronous callbacks (and (A)) correctly, the constraint on FP_C is that it is necessary to use parallel composition of the events corresponding to a particular callback `b` with other events, including the trigger `t` of `b`, according to the scheme

$FP_C = \beta_1\ op_1$ `?m`$\uparrow$ `;` β_2 `; !t`$\uparrow$ `; ((?t`$\downarrow$ `;` β_3 `; !m`$\downarrow$ $op_2\ \beta_4$`) | ?b)` (B1)

where β_i is composed of behavior protocols satisfying the constraint A connected via arbitrary behavior protocol operators except the consent (∇), and op_i is again either `;` or `+`. Specifically, a violation of the constraint would be to use explicit sequencing of events like in `?m`$\uparrow$ `; !t`$\uparrow$ `; ?t`$\downarrow$ `; ?b`$\uparrow$ `; !m`$\downarrow$ `; !b`$\downarrow$ (Tab. 1b), since an asynchronous callback runs in a different thread than the trigger and therefore unpredictable thread scheduling has to be considered. An example of a frame protocol that satisfies the constraint is

FP'_{Cb} = `?m1 ; ?m2`$\uparrow$ `; !t`$\uparrow$ `; ((?t`$\downarrow$ `; !m2`$\downarrow$ `; ?m3{!r3}) | ?b)` (EX-B1)

(C) To express autonomous activities on required interfaces (and (B)) correctly, the constraint is that it is also necessary to use parallel composition (as in (B)), since such activities are performed by C's inner threads and thus non-deterministic scheduling of the threads has to be considered. Specifically, the events corresponding to a particular autonomous activity `a` have to be composed via the and-parallel operator with other events that can occur after the start of the inner thread (in method `s`). Thus, when involving autonomous activities, FP_C has to comply with the scheme

$FP_C = \gamma_1\ op_1$ `?s`$\uparrow$ `; ((!s`$\downarrow$ `;` γ_2`) | !a)` (C1)

where γ_i is composed of behavior protocols satisfying the constraint B connected via arbitrary behavior protocol operators except the consent (∇). For example, the frame protocol $FP'_C = \gamma_1 op_1$ `?s ;` γ_2 `; !a`$\uparrow$ `;` γ_3 `; ?a`$\downarrow$ `;` γ_4 is not valid, since the events for the autonomous activity `a` are not allowed to happen before the call to `s` returns (i.e. before `!s`$\downarrow$ occurs). An example of a frame protocol that satisfies the constraint is

FP'_{Cc} = `?m1 ; ?s`$\uparrow$ `; ((!s`$\downarrow$ `; (?m2 + ?m3{!r3})) | !a)` (EX-C1)

In summary, to satisfy the constraints, a frame protocol has to be constructed in a hierarchical manner, with synchronous reactions and synchronous callbacks (compliant to the constraint A) lower than asynchronous callbacks (compliant to B), and with autonomous activities (compliant to C) at the top.

3.1 Calling & Trigger Protocol

An important question is whether from a FP_C (and frame protocols of other components at the same level of nesting as C) satisfying the constraints A, B, and C an EP_C^{call} can be derived such that it would be a valid model of behavior of C's environment; i.e., whether it suffices to model precisely only the interleaving of events on C's provided interfaces when callbacks and autonomous activities are considered. To answer this question, it is sufficient to consider the possible meanings of an event on a required interface in the frame protocol FP_C satisfying the constraints; such an event can be:

(1) A synchronous reaction `r` to a call on a provided interface, when `r` is not a trigger of a callback.

(2) An autonomous activity `a` on a required interface, when `a` is not a trigger of a callback.

(3) A trigger `t` of a callback `b` (either synchronous or asynchronous).

In cases (1) and (2), it is appropriate to model `r`, resp. `a`, implicitly (as in E1), since it has no relationship with any event on C's provided interfaces. On the other hand, a trigger `t` of a callback `b` (case 3) cannot be modeled implicitly, since `b` can be executed by E only after C invokes `t` - if `t` were modeled implicitly, then E could execute `b` even before `t` was invoked by the component.

Therefore, the answer to the question of sufficiency of the constraints is that the environment protocol based on a calling protocol (EP_C^{call}) is not a valid model of E's behavior if the interaction between C and E involves callbacks, since triggers of callbacks are modeled implicitly in EP_C^{call} - precise interleaving of a callback and its trigger has to be preserved in a valid model of E's behavior.

As a solution to this problem, we propose to define the environment protocol of a component C on the basis of a *calling & trigger protocol* that models a precise interleaving of the events on C's provided interfaces (including callbacks) and triggers of callbacks. In principle, the environment protocol takes the form

EP_C^{trig} = `<calling & trigger protocol> | ?m1* | ... |` (E2)
`| ?m1* | ?m2* | ... | ?mN*`

where `m1`, ..., `mN` are all the methods of the C's required interfaces except triggers of callbacks. Compliance of EP_C^{trig} with FP_C holds for similar reasons like in case of EP_C^{call} (end of Sect. 2.1) - note that although an environment modeled by EP_C^{trig} can accept triggers of callbacks from a component C only at particular moments of time, C will not invoke any trigger at an inappropriate time, since frame protocols of C and components cooperating with C at the same level of nesting are assumed to be compliant.

An environment protocol based on a calling & trigger protocol for (A1) has to comply with the scheme

$$EP_C^{trig} = \texttt{(}\alpha^{-1}_{1\_prov}\ op_1\ \texttt{!m\{}\alpha^{-1}_{2\_prov}\ op_2\ \texttt{?t\{!b\}}\ op_3\ \alpha^{-1}_{3\_prov}\texttt{\}}\ op_4\ op_4\ \alpha^{-1}_{4\_prov}\texttt{)}\ \texttt{|}\ \alpha^{-1}_{1\_req}\ \texttt{|}\ \ldots\ \texttt{|}\ \alpha^{-1}_{4\_req} \quad \text{(A2)}$$

where $\alpha^{-1}_{i\_prov}$ denotes the events on provided interfaces from α_i^{-1} and $\alpha^{-1}_{i\_req}$ denotes the events on required interfaces from α_i^{-1} (α_i^{-1} contains the events from α_i with `?` replaced by `!` and vice versa). For illustration, the proper environment protocol for (EX-A1) is $EP_{Ca}^{trig'}$ = `(!m1 ; !m2{?t{!b}} ; !m3) | ?r1 | ?r2 | ?r3`.

Similarly, an environment protocol for (B1) has to comply with the scheme

$$EP_C^{trig} = \texttt{(}\beta^{-1}_{1\_prov}\ op_1\ \texttt{!m}\uparrow\ \texttt{;}\ \beta^{-1}_{2\_prov}\ \texttt{;}\ \texttt{?t}\uparrow\ \texttt{;}\ \texttt{((!t}\downarrow\ \texttt{;}\ \beta^{-1}_{3\_prov}\ \texttt{;}\ \texttt{;}\ \texttt{?m}\downarrow\ op_2\ \beta^{-1}_{4\_prov}\texttt{)}\ \texttt{|}\ \texttt{!b))}\ \texttt{|}\ \beta^{-1}_{1\_req}\ \texttt{|}\ \ldots\ \texttt{|}\ \beta^{-1}_{4\_req}, \quad \text{(B2)}$$

while an environment protocol for (C1) has to comply with the scheme

$$EP_C^{trig} = \texttt{((}\gamma^{-1}_{1\_prov}\ op_1\ \texttt{!s}\ \texttt{;}\ \gamma^{-1}_{2\_prov}\texttt{)}\ \texttt{|}\ \texttt{?a)}\ \texttt{|}\ \gamma^{-1}_{1\_req}\ \texttt{|}\ \gamma^{-1}_{2\_req}. \quad \text{(C2)}$$

The proper environment protocol for (EX-B1) is $EP_{Cb}^{trig'}$ = `(!m1 ; !m2`$\uparrow$ `; ?t`$\uparrow$ `; ((!t`$\downarrow$ `; ?m2`$\downarrow$ `; !m3) | !b)) | ?r3`, while the proper environment protocol for (EX-C1) is $EP_{Cc}^{trig'}$ = `(!m1 ; !s`$\uparrow$ `; ((?s`$\downarrow$ `; (!m2 + !m3)) | !a)) | ?r3`.

3.2 Construction of Calling & Trigger Protocol

The algorithm for construction of a calling & trigger protocol (CTP) is based on the syntactical algorithm for derivation of a context protocol that was presented in [15] - the main difference is the newly added support for callbacks and autonomous activities. Only the basic idea is described here, i.e. technical details are omitted.

In general, the algorithm accepts frame protocols of all components (primitive and composite) in the given application and bindings between the components as an input, and its output are CTPs for all primitive components in the application. The frame protocols have to be augmented with identification of events that correspond to triggers for callbacks and autonomous activities.

The algorithm works in a recursive way: when executed on a specific composite component C, it computes CTP_{Ci} for each of its sub-components C_1, ..., C_N, and then applies itself recursively on each C_i.

More specifically, the following steps have to be performed to compute the calling & trigger protocol CTP_{Ck} of C_k, a sub-component of C:

1) A directed graph G of bindings between C and the sub-components of C is constructed and then pruned to form a sub-graph G_{Ck} that contains only the paths involving C_k. The sub-graph G_{Ck} contains a node N_C corresponding to C and a node N_{Ci} for each sub-component C_i of C; in particular, it contains a node N_{Ck} for C_k.

2) An intermediate version IP_{Ck} of CTP_{Ck} is constructed via a syntactical expansion of method call shortcuts during traversal of G_{Ck} in a DFS manner. The traversal consists of two phases - (i) processing synchronous reactions and autonomous activities on required interfaces, and (ii) processing callbacks. Technically, the first phase starts at N_C with CTP_C of C (inverted frame protocol is

used for the top-level composite component) and ends when all the edges on all paths between N_C and N_{Ck} are processed (cycles are ignored in this phase); the second phase starts at C_k and processes all cycles involving C_k. When processing a specific edge E_{lm}, which connects nodes N_{Cl} and N_{Cm} (for C_l and C_m), in the first phase, the current version IP_{Ck}^{lm} (computed prior to processing of E_{lm}) of IP_{Ck} is expanded in the following way: assuming that a required interface R_l of C_l is bound to a provided interface P_m of C_m, each method call shortcut on R_l in IP_{Ck}^{lm} is expanded to the corresponding method body defined in the frame protocol of C_m.

For example, if IP_{Ck}^{lm} contains "`...; !Rl.m1 ; !Rl.m2 ;...`" and the frame protocol of C_m contains "`...; ?Pm.m1{prot1} ; ?Pm.m2{prot2 + prot3} ; ...`", the result of one step of expansion has the form "`...; prot1 ; (prot2 + prot3) ;...`".

3) CTP_{Ck} is derived from IP_{Ck} by dropping (i) all the events related to other sub-components of C and (ii) all events on the required interfaces of C_k with the exception of triggers for callbacks, which have to be preserved.

In general, these three steps have to be performed for each sub-component of each composite component in the given component application in order to get a calling & trigger protocol for each primitive component.

4 Tools and Experiments

In order to show the benefits of use of the calling & trigger protocol-based approach instead of a context protocol or an inverted frame protocol, we have implemented construction of a context protocol (via consent composition) and a calling & trigger protocol (Sect. 3.2), and performed several experiments.

	Inverted frame protocol-based EP	Context protocol-based EP	Calling & trigger protocol-based EP
Time to compute EP	0 s	3 s	0.1 s
Total time (EP + JPF)	n/a	1102 s	1095 s
Total memory	> 2048 MB	762 MB	748 MB

Table 3. Results for the `Store` component

Our implementation of construction of a calling & trigger protocol and a context protocol does not depend on a specific component system, i.e. it can be used with any component system that supports formal behavior specification via behavior protocols (currently SOFA [4] and Fractal [1]). Moreover, the automated environment generator for JPF (EnvGen for JPF) [13] is available in both SOFA and Fractal versions, and thus we provide a complete JPF-based toolset for checking Java implementation of isolated SOFA or Fractal primitive components against the following properties: obeying of a frame protocol by the component's Java code [17] and all the properties supported by JPF out of the box (e.g. deadlocks and assertion violations).

	Inverted frame protocol-based EP	Context protocol-based EP	Calling & trigger protocol-based EP
Time to compute EP	0 s	2 s	0.5 s
Total time (EP + JPF)	n/a	n/a	485 s
Total memory	> 2048 MB	> 2048 MB	412 MB

Table 4. Results for the `ValidityChecker` component

We have performed several experiments on the `Store` component (Sect. 2.1) and the `ValidityChecker` component, which forms a part of the demo component application developed in the CRE project [1] - frame protocol, context protocol and calling & trigger protocol of `ValidityChecker` are in the appendix. For each experiment, we measured the following characteristics: time needed to compute a particular environment protocol, total time (computation of EP and JPF checking) and total memory; the value "> 2048 MB" for total memory means that JPF run out of available memory (2 GB) - total time is set to "n/a" in such a case.

Results of experiments (in Tab. 3 and Tab. 4) show that (i) construction of a calling & trigger protocol takes less time and memory than construction of a context protocol for these two components and (ii) total time and memory of environment's behavior model construction, environment generation and checking with JPF (against obeying of a frame protocol, deadlocks and race conditions) are the lowest if the calling & trigger protocol-based approach is used. Time needed to compute EP^{ctx} of both `Store` and `ValidityChecker` is also quite low, since frame protocols of other components bound to them (in the particular applications) do not involve very high level of parallelism and thus state explosion did not occur. The main result is that the whole process of environment construction and, above all, JPF checking has a lower time and space complexity for calling & trigger protocol than if the other approaches are used.

5 Evaluation and Related work

In general, our experiments confirm that although EP_C^{trig} for a component C specifies an "additional" parallelism (a parallel operator for each method of the C's required interfaces), the size of the JPF state space in checking C with an environment modeled by EP_C^{trig} is not increased (i.e. state explosion does not occur because of that), since the "additional" parallelism is not reflected in the environment's Java code explicitly via additional threads - the environment only has to be prepared to accept the call of any method from C (except triggers of callbacks) at any time and in parallel with other activities. On the contrary, modeling environment by EP_C^{trig} has the benefit of low time and space complexity (i) of construction of the model with respect to use of EP_C^{ctx}, and (ii) of JPF checking of component's Java code with respect to the use of EP_C^{inv}.

There are many other approaches to modeling behavior of software components and their environment that can be used to perform compositional verification of component-based applications (e.g. [9], [10], [5] and [12]); in particular, [9] and [10] do so on the basis of the assume-guarantee paradigm. However, to our knowledge, none of them supports independent constructs for the following atomic events explicitly in the modeling language: acceptance of a method invocation (`?i.m`$\uparrow$ in behavior protocols), emitting a method invocation (`!i.m`$\uparrow$), acceptance of a return from a method (`?i.m`$\downarrow$), and emitting a return from a method (`!i.m`$\downarrow$). Process algebra-based approaches ([9], [5]) typically support input (acceptance) and output (emit) actions explicitly in the modeling language, while transition systems-based approaches (e.g. [10] and [12]) support general events. In any case, it is possible to distinguish the events via usage of different names (e.g. event names `m1_invoke`, resp. `m1_return`, for invocation of `m1`, resp. for return from the method); however, an automated composition checking may fail even for compliant behavior specifications in such a case, since the developer of each of them can choose a different naming scheme (e.g. `m1_invoke` versus `m1`$\uparrow$). We believe that a formalism for modeling component behavior should support all the four types of atomic events, since:

(a) independent constructs for method invocation and return from a method are necessary for proper modeling of callbacks and autonomous activities, and

(b) independent input and output actions are necessary for compliance checking, i.e. for checking the absence of communication errors between components.

Program model checking of open systems (isolated software components, device drivers, etc) typically involves construction of an "artificial" environment - an open system subject to checking and its environment then form a closed system (a complete program). The environment typically has the form of a program, as in our approach [14] and in [10], where the environment is defined in Java, or in SLAM/SDV [2], where the model of the windows kernel (environment for device drivers) is defined in the C language. In general, each approach to model checking of open software systems involves a custom tool or algorithm for construction of the environment, since each program model checker features a unique combination of API and input modeling language (i.e. different combination than the other program model checkers).

As for automated construction of the model of environment's behavior, one recent approach [7] is based on the L^* algorithm for incremental learning of regular languages. The basic idea of this approach is to iteratively refine an initial assumption about behavior of the environment for a component subject to checking. At each step of the iteration, model checking is used to check whether the component satisfies the property, and if not, the assumption is modified according to the counterexample. The iteration terminates when the component satisfies the given property in the environment modeled by the assumption. An advantage of our approach over [7] is lower time and memory complexity, since use of model checking is not needed for construction of EP^{trig}.

6 Conclusion

In our former work, we introduced two specific approaches to modeling of environment's behavior: inverted frame protocol and context protocol. However, JPF checking of a component with the environment determined by any of these modeling approaches is prone to state explosion for the following reasons: (i) Java code of the environment is complex, since it has to ensure proper interleaving of invocation and return events on the component's provided and required interfaces, (ii) for the context protocol, the algorithm for its construction involves model checking, while for the inverted frame protocol, the environment involves high level of parallelism. To address the problem of state explosion, in [16] we proposed to use a model of environment's behavior based on the calling protocol. Since the calling protocol-based approach models precisely only the events on component's provided interfaces, it does not allow to express C-E patterns properly in general (it is an overapproximation of the desired behavior).

Therefore, in this paper we proposed a slightly modified idea - calling & trigger protocol, which models precise interleaving of events on provided interfaces and triggers of callbacks, and the "other events" models implicitly, similar to [16] with no threat of state explosion. The key idea is to impose certain constraints on the frame protocol of a component in terms of interleaving of C-E events with other events and to express the constraints via syntactical patterns the frame protocol has to follow, and then, if the constrains are satisfied, derive in an automated way the calling & trigger protocol. The experiments confirm that the idea is viable.

As a future work, we plan to create a tool for automated recognition of those component frame protocols that do not satisfy the constraints and to integrate it into the SOFA runtime environment.

Acknowledgments. This work was partially supported by the Grant Agency of the Czech Republic (project number 201/06/0770).

References

1. Adamek, J., Bures, T., Jezek, P., Kofron, J., Mencl, V., Parizek, P., Plasil, F.: Component Reliability Extensions for Fractal Component Model, `http://kraken.cs.cas.cz/ft/public/public_index.phtml`, 2006.
2. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., Ustuner, A.: Thorough Static Analysis of Device Drivers, Proceedings of EuroSys 2006, ACM Press
3. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.B.: The FRACTAL component model and its support in Java, Softw. Pract. Exper., 36(11-12), 2006

4. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proceedings of SERA 2006, IEEE CS
5. Brim, L., Cerna, I., Varekova, P., Zimmerova, B.: Component-interaction Automata as a Verification-oriented Component-based System Specification, Proceedings of SAVCBS 2005, ACM Press
6. Clarke, E. M., Long, D. E., McMillan, K. L.: Compositional Model Checking, Proceedings of LICS'89, IEEE CS
7. Cobleigh, J. M., Giannakopoulou, D., Pasareanu, C. S.: Learning Assumptions for Compositional Verification, Proceedings of 9th TACAS, LNCS, vol. 2619, 2003
8. CoCoME, `http://agrausch.informatik.uni-kl.de/CoCoME`
9. de Alfaro, L., Henzinger, T. A.: Interface Automata, Proceedings of 8th European Software Engineering Conference, ACM Press, 2001
10. Giannakopoulou, D., Pasareanu, C. S., Cobleigh, J. M.: Assume-guarantee Verification of Source Code with Design-Level Assumptions, Proceedings of 26th International Conference on Software Engineering (ICSE), 2004
11. Mach, M., Plasil, F., Kofron, J.: Behavior Protocol Verification: Fighting State Explosion, International Journal of Computer and Information Science, 6(2005)
12. Ostroff, J.: Composition and Refinement of Discrete Real-Time Systems, ACM Transactions on Software Engineering and Methodology, 8(1), 1999
13. Parizek, P.: Environment Generator for Java PathFinder, `http://dsrg.mff.cuni.cz/projects/envgen`
14. Parizek, P., Plasil, F.: Specification and Generation of Environment for Model Checking of Software Components, Proceedings of FESCA 2006, ENTCS, 176(2)
15. Parizek, P., Plasil, F.: Modeling Environment for Component Model Checking from Hierarchical Architecture, Proceedings of FACS'06, ENTCS, vol. 182
16. Parizek, P., Plasil, F.: Partial Verification of Software Components: Heuristics for Environment Construction, Proc. of 33rd EUROMICRO SEAA, IEEE CS, 2007
17. Parizek, P., Plasil, F., Kofron, J.: Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker, Proceedings of SEW'06, IEEE CS
18. Pasareanu, C. S., Dwyer, M., Huth, M.: Assume-guarantee model checking of software: A comparative case study, Proceedings of the 6th SPIN workshop, LNCS, vol. 1680, 1999
19. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, 28(11), 2002
20. Tkachuk, O., Dwyer, M. B., Pasareanu, C. S.: Automated Environment Generation for Software Model Checking, Proceedings of ASE 2003, IEEE CS
21. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs, Automated Software Engineering Journal, vol. 10, no. 2, 2003

Appendix

$FP_{ValidityChecker} =$

```
(
  ?IToken.SetEvidence
  |
  ?IToken.SetValidity
```

```
    |
    (
      ?IToken.SetAccountCredentials {
        !ICustomCallback.SetAccountCredentials
      }
      +
      NULL
    )
  )
  ;
  ?ILifetimeController.Start^ ; !ITimer.SetTimeout^
  ;
  (
    (
      ?Timer.SetTimeout$ ; !ILifetimeController.Start$
      ;
      (
        ?IToken.InvalidateAndSave {
          !ITimer.CancelTimeouts;
          (!ICustomCallback.InvalidatingToken + NULL);
          !ITokenCallback.TokenInvalidated
        }*
        |
        ?IToken.InvalidateAndSave {
          !ITimer.CancelTimeouts;
          (!ICustomCallback.InvalidatingToken + NULL);
          !ITokenCallback.TokenInvalidated
        }*
      )
    )
    |
    ?ITimerCallback.Timeout {
      (!ICustomCallback.InvalidatingToken + NULL);
      !ITokenCallback.TokenInvalidated
    }*
  )
```

$EP^{inv}_{ValidityChecker} = EP^{ctx}_{ValidityChecker} =$

```
  (
    !IToken.SetEvidence
    |
    !IToken.SetValidity
    |
    (
      !IToken.SetAccountCredentials {
        ?ICustomCallback.SetAccountCredentials
      }
      +
      NULL
    )
  )
  ;
  !ILifetimeController.Start^ ; ?ITimer.SetTimeout^
  ;
  (
    (
      !Timer.SetTimeout$ ; ?ILifetimeController.Start$
      ;
      (
        !IToken.InvalidateAndSave {
          ?ITimer.CancelTimeouts;
          (?ICustomCallback.InvalidatingToken + NULL);
          ?ITokenCallback.TokenInvalidated
        }*
        |
        !IToken.InvalidateAndSave {
          ?ITimer.CancelTimeouts;
          (?ICustomCallback.InvalidatingToken + NULL);
          ?ITokenCallback.TokenInvalidated
        }*
      )
    )
    |
    !ITimerCallback.Timeout {
      (?ICustomCallback.InvalidatingToken + NULL);
      ?ITokenCallback.TokenInvalidated
    }*
  )
```

$EP^{trig}_{ValidityChecker} =$

```
  (
    (
      !IToken.SetEvidence
      |
      !IToken.SetValidity
      |
      (
        !IToken.SetAccountCredentials
        +
        NULL
      )
    )
    ;
    !ILifetimeController.Start^ ; ?ITimer.SetTimeout^
    ;
    (
      (
        !Timer.SetTimeout$ ; ?ILifetimeController.Start$
        ;
        (
          !IToken.InvalidateAndSave*
          |
          !IToken.InvalidateAndSave*
        )
      )
      |
      !ITimerCallback.Timeout*
    )
  )
  |
  ?ICustomCallback.SetAccountCredentials*
  |
  ?ITimer.CancelTimeouts*
  |
  ?ITimer.CancelTimeouts*
  |
  ?ICustomCallback.InvalidatingToken*
  |
  ?ICustomCallback.InvalidatingToken*
  |
  ?ICustomCallback.InvalidatingToken*
  |
  ?ITokenCallback.TokenInvalidated*
  |
  ?ITokenCallback.TokenInvalidated*
  |
  ?ITokenCallback.TokenInvalidated*
```