Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 182 (2007) 139–153
www.elsevier.com/locate/entcs

Modeling Environment for Component Model Checking from Hierarchical Architecture

Pavel Parizek[a,1], Frantisek Plasil[a,b,1]

[a] *Department of Software Engineering*
Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
{parizek, plasil} @ nenya.ms.mff.cuni.cz

[b] *Institute of Computer Science*
Academy of Sciences of the Czech Republic
Prague, Czech Republic
plasil @ cs.cas.cz

Abstract

Application of model checking to isolated software components is not directly possible because a component does not form a complete program - the problem of missing environment occurs. A solution is to create an environment of some form for the component subject to model checking. As the most general environment can cause model checking of the component to be infeasible, we model the environment on the basis of a particular context the component is to be used in. More specifically, our approach exploits hierarchical component architecture and component behavior specification defined via behavior protocols, all that provided in ADL. This way, the environment represents the behavior of the rest of the particular application with respect to the target component. We present an algorithm for computing the model of environment's behavior that is based on syntactical expansion and substitution of behavior protocols.

Keywords: Software components, behavior protocols, environment for model checking, hierarchical component architecture

1 Introduction

Various methods of formal verification have already proven to be useful for finding errors in large and complex software systems, and particularly in critical systems, thus helping increase reliability of such systems. At present, one of the most popular approaches to verification of software systems is model checking [3], which is an algorithmic technique for checking whether a finite model of a target system satisfies a certain property. Typically, the model has the form of a finite labeled transition system and the property can be expressed as a temporal logic expression

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

doi:10.1016/j.entcs.2006.09.036

(LTL, CTL). Checking whether a property is satisfied in the model is based on an exhaustive traversal of the state space determined by the model. This way, model checking can help to find concurrency errors like deadlocks, which are very subtle and quite hard to discover with traditional approaches such as testing. However, the main advantage of model checking - traversal of the complete state space (i.e. checking of the property in each state) - is also its main weakness. Especially in case of more complex software systems, the state space may be large enough to make model checking of a system not feasible; this is well-known as the *state explosion problem*.

A completely different approach to building more reliable software systems is to decompose large and complex systems into smaller and well-defined units - software components. Typically, components are considered to be entities with well-defined provided (server) and required (client) interfaces, and in some cases also with formally specified behavior. A component-based application is a collection of individual components, which are interconnected via well-defined bindings between their interfaces.

Components that have no externally observable internal structure, while having real implementation in a certain programming language, are called *primitive components*. Components containing nested subcomponents, i.e. components with observable internal structure, are called *composite components*. The structure of a composite component, commonly referred to as *component architecture*, is typically defined in an Architecture Description Language (ADL) [2][7][8]. Usually, definition of a composite component in an ADL specifies also the external provided and required interfaces of all components, bindings between the component and its subcomponents, and optionally component behavior (e.g. in an LTS-based formalism).

An example of a composite component, which will be used to illustrate the ideas presented throughout the rest of the paper, is depicted on Fig. 1. The component `DBServer` provides the `db` interface of type `IDatabase` and contains four primitive subcomponents - `Database`, `Logger`, `Transaction Manager`, and `Backup Scheduler`. The `db` interface of the `DBServer` component is implemented by delegation to the `Database` subcomponent. The `Logger` and `Transaction Manager` subcomponents are bound to the required interfaces of the `Database` component, and the `Transaction Manager` component is bound also to the required interface of the `Backup Scheduler` component. In the rest of the paper, we will be interested especially in the `Transaction Manager` component, which provides the `start`, `stop`, `begin`, `commit` and `abort` methods on its provided interface `tm` of type `ITxMngr`, and the `backup` method on its provided interface `bk` of type `IBackup`.

1.1 Problem of Missing Environment

A viable approach is the application of model checking to individual software components, for example, in order to verify that the component's implementation satisfies a formal specification of the component's behavior. As an individual component obviously generates a smaller state space than the whole application, the problem of state explosion is also mitigated this way, at least partially.

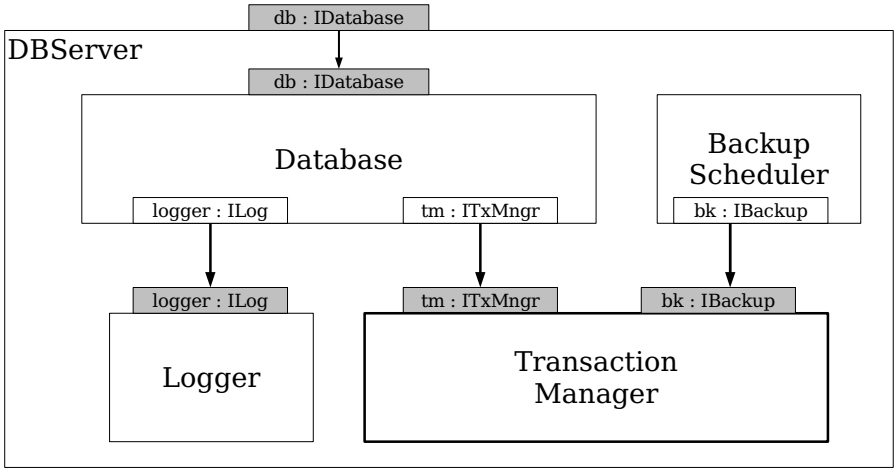

Fig. 1. Architecture of the `DBServer` component

However, considering only primitive components, the problem with model checking of these components is that they are not complete programs (e.g. with `main` method) - and model checkers typically analyze only complete programs. This triggers the *problem of missing environment*. An obvious solution is to create an environment of some form for each primitive component subject to model checking, and then separately check the complete programs, each composed of a primitive component and its environment.

Such an environment has to fulfill the following key requirements:

- It should be created in a way that minimizes the state space size of the program composed from the component and its environment, while at the same time, it should be complex enough to exercise the target component under all reasonable behaviors (sequences and parallel interleavings of method calls) and all combinations of input values.
- It should allow the model checker to search for concurrency errors; this is typically reflected by calling methods of the component by more threads of control.
- It should force the model checker to check all the control flow paths in the component's code; this is usually addressed by calling each method of the component several times with different combinations of parameter values - particular combination for a method invocation being selected non-deterministically via means provided by the model checker.

An environment for `Transaction Manager` (from Fig. 1) that fulfills all three requirements could take the form depicted in Fig. 2 (only fragments of its Java code are presented). From that it is clear that manual construction of the environment is tedious and error-prone process. Therefore, we aim at creating a tool that would generate the environment in an automated way, i.e. an environment generator.

As an input, the environment generator will get the specification of a component's environment, which has to determine all behaviors and combinations of input values. Having the proper environment specification, the tool is able to produce a reasonable environment for component model checking, which fulfills all the requirements stated above.

```
public class EnvDbThread extends Thread
{
  ITxMngr tm;

  public void run()
  {
    String id = tm.begin(getRandomString());
    if (getRandomBool()) tm.commit(id);
    else tm.abort(id);
    ...
  }
}

public class EnvBkThread extends Thread
{
  IBackup bk;

  public void run()
  {
    bk.backup(); ...
  }
}

public static void main(String[] args)
{
  TransactionMngrImpl tm = new TransactionMngrImpl();

  tm.start();

  new EnvDbThread(tm).start();
  new EnvDbThread(tm).start();
  new EnvBkThread(tm).start();

  tm.stop();
}
```

Fig. 2. Fragments of Java code of environment for the `Transaction Manager` component

The specification itself may be divided into (i) a model of the environment's behavior and (ii) a definition of possible combinations of input values (i.e. method

parameters). All of this can be provided manually by the user or retrieved, for example, from the ADL specification of the whole component-based application. In this text, we focus on modeling of the environment's behavior (our current approach to definition of possible combinations of input values is described in [9]).

1.2 Goals and Structure of the Paper

The paper aims at addressing the problem of modeling the environment for model checking of primitive software components that have their behavior specified in the formalism of behavior protocols [12]. The main goal is to present our approach to modeling of the environment's behavior, which exploits the definition of a component's architecture and specification of the component behavior via behavior protocols provided in ADL.

The remainder of the paper is organized as follows. Sect. 2 provides an overview of behavior protocols. Sect. 3 presents the key contribution - our solution to computing the model of environment's behavior from (i) the graph of bindings between components in the architecture and (ii) the behavior specifications of all the components (defined via behavior protocols) in the architecture. The rest of the paper contains evaluation, related work and a conclusion.

2 Behavior Protocols

For specification and modeling of behavior of software components, we use the formalism of behavior protocols.

A behavior protocol is an expression that specifies the behavior of a software component in terms of specific atomic events on the component's provided and required interfaces, those events being accepted and emitted method call requests and responses. Each behavior protocol defines a possibly infinite set of traces, where each trace is a finite sequence of atomic events - we use $L(prot)$ to denote the set of traces specified by a protocol *prot*. The semantics of a behavior protocol is defined in terms of labeled transition system (LTS), with transitions labeled by atomic events.

Syntactically, a behavior protocol reminds a regular expression, with a set of atomic actions working as the underlying alphabet. Each atomic event has the following syntax: `<prefix><interface>.<method><suffix>`. The prefix `?` denotes an accept event, the prefix `!` denotes an emit event, the suffix ↑ denotes a request (i.e. a method call), and the suffix ↓ denotes a response (i.e. return from a method). Several shortcuts, which make the protocols more readable, are also defined. For example, an expression of the form `?i.m{prot}` is a shortcut for the protocol `?i.m↑;prot;!i.m↓`, and an expression of the form `?i.m` is a shortcut for the protocol `?i.m↑;!i.m↓`. The `NULL` keyword denotes an empty protocol.

In addition to standard operators `;` (sequence), `+` (alternative), and `*` (repetition), behavior protocols provide the and-parallel operator `|`, which generates all the possible interleavings of event traces defined by its operands, and the or-parallel operator `||` (`p || q` stands for `p + q + (p | q)`).

The component's *frame protocol* [12] describes the external behavior of the component by defining all the valid sequences of events (i.e. traces) on the component's external interfaces. For composite components, the *architecture protocol* describes the composed behavior of all subcomponents at the first level of nesting; it is generated as a parallel composition of frame protocols of the subcomponents.

The frame protocol of `Transaction Manager` (Sect. 1) is

```
( ?tm.start ; ?tm.begin* ; (?tm.begin* | ?tm.commit* |
?tm.abort*) ; ?tm.stop ) | ?bk.backup*
```

It is a parallel composition of two subprotocols. The first of them specifies that the component should accept finite number of calls of `backup` on the `bk` interface. The second subprotocol states that the component has to accept call of `start` on its `tm` interface and then a finite number of calls of `begin` on `tm`, then it should accept calls of `begin`, `commit` and `abort` on `tm` in parallel, and finally it should accept the call of `stop` on `tm`.

The frame protocol of `Database` might be

```
?db.start{!logger.start ; !tm.start} ;
(
  ?db.add{!tm.begin ; (!tm.commit + !tm.abort)}
  +
  ?db.get{!tm.begin ; (!tm.commit + !tm.abort)}
  +
  ?db.remove{!tm.begin ; (!tm.commit + !tm.abort)}
)* ;
?db.stop{!logger.stop ; !tm.stop},
```

the frame protocol of `DBServer` might be

```
?db.start ; (?db.add + ?db.get + ?db.remove)* ; ?db.stop,
```

and the frame protocol of `Backup Scheduler` might be `!bk.backup*`.

An advantage of using behavior protocols for specification of component's behavior is the possibility to check whether the components equipped with frame protocols are behaviorally compliant, i.e. whether the components communicate without errors. We distinguish between (i) the *horizontal compliance* of components at the same level of nesting and (ii) the *vertical compliance* of a frame of a composite component with the underlying architecture (expressed by the architecture protocol). Nevertheless, checking of behavior compliance makes sense only under the assumption that the implementation of each primitive component satisfies its frame protocol (we say that the component *obeys* its frame protocol). This holds only if the component accepts/issues only such method-call related event sequences on its external provided and required interfaces that are specified by the component's frame protocol. An obvious approach to checking whether a component obeys its frame protocol is to use code model checking; for that purpose we have a tool [10] that accepts only complete programs as input, and therefore we need to create an environment that, together with the component, makes a complete program accepted

by our tool.

3 Modeling the Environment with Behavior Protocols

As indicated in Sect. 1.1, a model of the environment's behavior has to be supplied as a part of the environment specification that is provided to an environment generator. The model of the environment's behavior should reflect the fact that the resulting environment has to represent the behavior of all other components that can possibly be bound to the target component. As an example, consider the component architecture on Fig. 1; the environment for `Transaction Manager` should represent at least the behavior of `Database` and `Backup Scheduler` with respect to `Transaction Manager`.

Our first solution to modeling of the environment's behavior, presented in [9], uses the *inverted frame protocol* [1] of the target component, which is constructed from the component's frame protocol by replacing all the accept events by emit events and vice versa. Such a model is the most general one, as the component's frame protocol specifies all the sequences of events the component can accept/issue on its external (provided and required) interfaces. A drawback of this solution is that the environment generated this way can be very complex, frequently making model checking of the program composed of the component and its environment suffer from state explosion. In [9] we presented an attempt to mitigate this drawback by designing heuristic transformations and approximations of the frame protocol that simplify the resulting environment to an extent that makes checking feasible. However, a problem with this approach is that the resulting environment exercises the target component only by a subset of the behaviors defined by the component's frame protocol; therefore, checking whether the component obeys its frame protocol is not exhaustive in such a case.

In order to solve this problem, we propose a new approach to modeling the environment's behavior on the basis of a particular context - in our case, an architecture the component is expected to be used in. More specifically, our approach exploits (i) the definition of the architecture the target component is a part of, and (ii) the behavior specification (defined as behavior protocols) of all the components that form the architecture. Here, the basic idea is to use *context protocol* of the target component, which specifies the actual use of the target component by the other components of the architecture (and vice versa), as the model of the environment's behavior.

Using the context protocol instead of the inverted frame protocol as the model of the environment's behavior is useful especially in the case, where a particular component-based application exploits only a subset of the functionality provided by the target component - the context protocol then specifies only a subset of behaviors determined by the inverted frame protocol, thus helping mitigate the problem of state explosion.

To illustrate the advantage of using the context protocol instead of the inverted frame protocol, consider again the architecture on Fig. 1, having the frame protocols

of the `Transaction Manager`, `Database`, and `Backup Scheduler` components as presented in Sect. 2. Since the frame protocol of `Database` effectively specifies call of `begin` on its required interface `tm` followed by an alternative between calls to `commit` and `abort` on `tm`, all that repeated for a finite number of times, then, despite the fact that the frame protocol of `Transaction Manager` specifies parallel calls of those methods, the context protocol for `Transaction Manager` is

```
( !tm.start ; (!tm.begin ; (!tm.commit + !tm.abort))* ; !tm.stop )
| !bk.backup*
```

Such a context protocol for `Transaction Manager` obviously specifies a subset of behaviors determined by the component's (inverted) frame protocol, and, therefore, model checking of `Transaction Manager` with the environment modeled by this context protocol will have lower time and space requirements, than if the inverted frame protocol was used for this purpose.

Notice also, that the behavior determined by the component's context protocol has to be a subset of behavior specified by the component's inverted frame protocol for the checking of a component against its frame protocol to work correctly; otherwise the model checker could report some "false errors" in addition to violations of the component's frame protocol by its implementation, since also the traces not allowed by the frame protocol will be defined in the context protocol (with corresponding behavior being encoded in the generated environment) in this case. In other words, for the inverted frame protocol IF_C of the component C and its context protocol CTX_C, the formula $L(CTX_C) \subseteq L(IF_C)$ must hold.

3.1 Computing the Model of Environment's Behavior

Technically, our approach is to compute a behavior protocol that models behavior of target component's environment from (i) frame protocols of the other components at the same level of nesting, (ii) the inverted frame protocol of the parent component and (iii) the bindings between component's interfaces; we denote the output, i.e. the model of the environment's behavior, to be the *environment protocol* of the target component. The ideal algorithm for this purpose is the one that fulfills the following two requirements:

- It should take only a fraction of time required by actual model checking of the target component, as the task of environment construction is only a prerequisite to the process of model checking, which has big time and space requirements on its own.
- The algorithm should be precise; i.e. the resulting environment protocol should represent exactly those behaviors that can be exercised on the target component by other components taking part in the particular architecture, i.e. it should specify the same behavior like the target component's context protocol. Representing a subset of those behaviors would prevent exhaustive model checking and representing a superset of those behaviors could possibly reduce efficiency of the checking (by increasing the state space size).

However, for the algorithm to have low time requirements (which is our top priority), it is necessary to make a compromise on the second requirement, as computing the environment protocol that specifies exactly the same behavior as the context protocol could be a very time- and space-consuming task for some inputs. In such cases, the algorithm should produce an environment protocol that is a superset of the context protocol (in terms of behavior specified by it) in an efficient way. Nevertheless, considering the inverted frame protocol IF_C of the component C, its context protocol CTX_C, and its environment protocol E_C, then the formula $L(CTX_C) \subseteq L(E_C) \subseteq L(IF_C)$ must hold. Consequently, the component's environment protocol will specify the same behavior as its inverted frame protocol in the worst case.

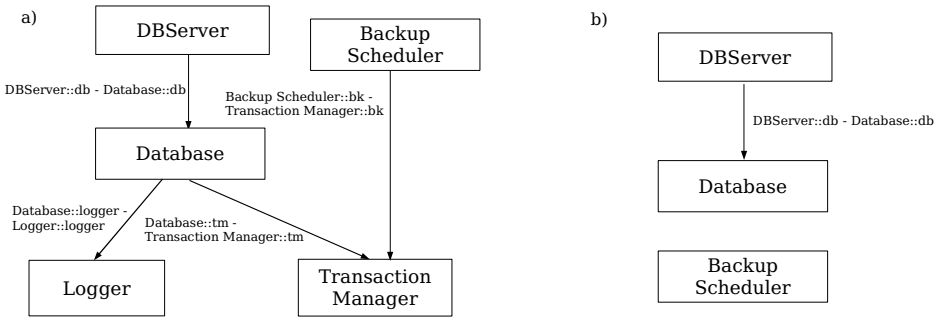

Fig. 3. a) graph of bindings between components; b) binding trees for the `Transaction Manager` component

3.1.1 Syntactical Approach

We designed an algorithm, which is based on syntactical expansion and substitution of (parts of) behavior protocols. Its input consists of the frame protocols of all the components in the architecture (except the target one) and the graph of the bindings between the components, and its output is the environment protocol of the target component. The algorithm is divided into three steps, described below.

The first step is the reduction of the graph of bindings to a subgraph that contains only the paths that start at a parent component or at a component with no provided interfaces (or with all its provided interfaces unbound) and end at a component that is bound to the target component. In fact, the subgraph is a set of acyclic graphs, which we call *binding trees* (despite the fact that some of them may actually be DAGs) - there is one binding tree for the parent component, if defined in the architecture, and one binding tree for each component with no provided interfaces (or with all its provided interfaces unbound). Considering the architecture on Fig. 1, the graph of bindings on Fig. 3a, and the `Transaction Manager` component as the target one, this step of the algorithm will produce a subgraph that is depicted at the Fig. 3b. The first binding tree of the subgraph consists of two nodes - the root node corresponding to the `DBServer` component,

and its child node corresponding to the `Database` component - and one edge that represents the binding between the two components, and the second binding tree consists of one node corresponding to the `Backup Scheduler` component.

In the second step, a part of the environment protocol is constructed for each binding tree via syntactical expansion of protocols during traversal of a tree in the DFS manner. The frame protocol of the component (or inverted frame protocol in case of a parent component) corresponding to the root node of a tree represents the initial version of the part of the environment protocol for the particular binding tree. Then, when backtracking over an edge from a node A to a node B (which is the parent of A) during DFS, all bindings between the required interfaces of the component C_B (represented by the node B) and provided interfaces of the component C_A (represented by the node A) are taken, and for each of these bindings all the calls on the corresponding required interface of C_B (as defined in its frame protocol) are replaced with reactions to those calls (as defined in the frame protocol of C_A) in the current version of the part of the environment protocol. If the frame protocol of C_A specifies two or more reactions to some specific method call that are connected via the and-parallel operator, it is necessary to use all these reactions together with the connecting and-parallel operators preserved for the purpose of replacing the corresponding call.

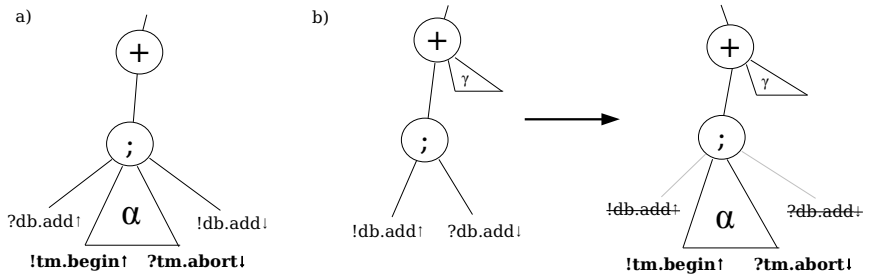

Fig. 4. a) parse tree for the frame protocol of `Database`; b) illustration of one step in construction of the environment protocol for `Transaction Manager` - replacement of call to `db.add` with a reaction to the call (specified in the frame protocol of `Database`)

For illustration of the syntactical expansion of protocols, consider the first binding tree on Fig. 3b and the `Transaction Manager` component as the target one. Then, when backtracking over the edge that represents the binding between `DBServer` and `Database`, the call `!db.add` will be expanded to a reaction to this call that is specified in the frame protocol of `Database`, i.e. to a subprotocol `!tm.begin ; (!tm.commit + !tm.abort)`, as depicted on Fig. 4 (the figure showing only fragments of parse trees of the protocols).

In the third step of the algorithm, parts of the environment protocol for all binding trees are connected via the and-parallel operator, thus forming the resulting environment protocol for the target component. Using the and-parallel operator is necessary because calls delegated from the parent component (if it exists) can be

performed in parallel with calls performed by components that have no provided interfaces (or have all of them unbound). Considering our example, there are two binding trees, one with the `DBServer` component as its root node and the second with the `Backup Scheduler` component as its root node; therefore, the two parts of the environment protocol that correspond to these binding trees will be connected with the and-parallel operator.

Finally, the environment protocol is simplified to contain only events that represent calls on the provided interfaces of the target component, as all other events are not relevant for modeling the environment of the target component and can be therefore safely ignored.

The output of our algorithm for the `Transaction Manager` component is depicted in Fig. 5. It is an environment protocol that specifies the same behavior as the component's context protocol presented in Sect. 3, i.e. both protocols specify the same set of event sequences. The presence of an alternative between subprotocols of the form `(!tm.begin ; (!tm.commit + !tm.abort))` is only a syntactical difference, which could be handled by preprocessing of some form before the environment is actually generated from the environment protocol. The reason for the environment protocol to have this form, not allowing parallel invocation of methods on the `tm` interface, is that the frame protocol of `DBServer` specifies no parallelism; calls on `db` specified in the inverted frame protocol of `DBServer` are replaced with reactions to those method calls that are specified in the frame protocol of `Database`, when the environment protocol is constructed.

```
(
  !tm.start ;
  (
    (!tm.begin ; (!tm.commit + !tm.abort))
    +
    (!tm.begin ; (!tm.commit + !tm.abort))
    +
    (!tm.begin ; (!tm.commit + !tm.abort))
  )* ;
  !tm.stop
)
|
!bk.backup*
```

Fig. 5. Environment protocol for the `Transaction Manager` component

Consequence of the environment protocol for `Transaction Manager` specifying only repetition of alternative calls on the `tm` interface (i.e. real usage of the component in the given architecture) is that the environment modeled by this environment protocol will determine smaller state space (of the program composed of the component and environment) than if the component's inverted frame protocol, which allows for parallel calls of these methods, is used for modeling of the environment's behavior.

4 Evaluation

As already mentioned in Sect. 3, an advantage of modeling the environment's behavior via environment protocol is that the environment protocol reflects the real usage of the target component in the specific architecture the component is used in, and, therefore, it will typically specify a subset of behaviors determined by the inverted frame protocol of the target component. On the other hand, a drawback of using the environment protocol is that checking of the component has to be performed again for each architecture the component is used in, since a different subset of behaviors defined by the component's frame protocol may be exploited in each component architecture. In any case, checking whether the target component obeys its frame protocol is exhaustive (with respect to the specific architecture), if the environment protocol is used - not like when the heuristic transformations of the inverted frame protocol are applied, which make checking of the component not exhaustive (although more feasible in most cases).

The algorithm for computing the environment protocol, which is described in Sect. 3.1.1, works well and produces expected results; its time and space requirements being fractional with respect to actual checking of the component. For example, the algorithm is able to detect that methods of the target component are called sequentially or alternatively in the given architecture, even though the component's frame protocol allows for parallel calls of the methods of the component (see the example in Sect. 3.1.1 for illustration). Nevertheless, it is hard to estimate the state space reduction achieved by our approach in general, as the level of reduction depends on each specific case (i.e. how the target component is used in the particular architecture); a systematic analysis of this issue is subject of our current research. We also have a proof-of-concept implementation that was tested on several examples, including the one presented in this paper.

However, our solution has also some drawbacks. One of them is that the syntax-based algorithm does not produce a correct environment protocol if the component is able to perform some calls on its required interfaces autonomously. Specifically, such calls will not be included in the resulting environment protocol, which is then incorrect because it will not force the environment to wait for these calls to happen (i.e. wait till the component issues the calls). The reason for not including autonomous calls on component's required interfaces in the environment protocol is that binding trees does not contain bindings of the target component's required interfaces to the provided interfaces of other components in the architecture.

Second problem of our approach is that syntactical expansion of protocols may not produce a correct result in cases, when the frame protocol of a certain component specifies more reactions to some method call that are connected via the sequence operator. In that case, it is generally not possible to decide, in an efficient way, which reaction is the appropriate one for the particular method call.

Third drawback of our solution, less significant than the first two, is that our algorithm may produce an environment protocol that specifies a superset of behaviors determined by the context protocol for some inputs. Nevertheless, one of our

design requirements was to develop an efficient algorithm with respect to both time and space, and this has been achieved with designing the algorithm to produce the environment protocol that specifies more behaviors than the context protocol in some cases.

5 Related work

The problem of model checking of isolated software components, which form a component-based application, can be seen as a variation of compositional model checking [4], whose basic idea is to (i) decompose a target system into several components, (ii) verify local properties of the components via model checking, and (iii) deduce global properties of the whole system from the local properties of the components. The key point of this approach is checking properties of a composition of a selected component with a model of its environment, instead of checking properties of the isolated component; by using an environment, it is guaranteed that the checked local properties are preserved also at the global level. The difference between compositional model checking and our approach is that the former aims at checking global properties of the whole program (or a set of processes) via checking local properties of individual components (or processes), while our approach aims at checking the properties specific to individual components (e.g. obeying of a frame protocol).

For verification of properties of software components, the assume-guarantee approach [6][11] is often used. The idea is to check a component only in such environments that satisfy certain assumptions typically provided by the user; we can say that the assumptions model the valid environments of a component subject to model checking. This way, the need to check the component in all possible environments (or in a universal environment), what is usually an infeasible task, is avoided. Application of model checking to a component with an environment satisfying a certain assumption then verifies whether the component satisfies the given property under the given assumption. If the model checker returns a positive answer, it is guaranteed that the component, when used in an environment that satisfies the specific assumption, must satisfy the given property in this environment. In order for the checking of a component to be of practical use, the assumptions should together model the real environment of the component (e.g. an architecture the component is to be used in). The most popular means for expressing the assumptions is the temporal logic (LTL), which is commonly used also for specification of the properties.

However, in our specific case, we use the environment protocol as the assumption about the environment for model checking of components. This way, we do not have to check whether such an assumption holds for the environment actually used, as the environment is generated on the basis of the environment protocol, and therefore the assumption represented by the environment protocol holds trivially. Typical application of the assume-guarantee paradigm also requires the assumptions to be manually created by the user; in our case, we have to manually define the frame

protocols of all the components in the architecture - the environment protocol is then automatically computed from these frame protocols and bindings between components (this info is stored in the corresponding ADL file). We are aware of only one automatic approach to generating the assumptions for compositional verification, which is based on incremental learning [5].

6 Summary

Direct model checking of isolated software components is typically not possible because a component does not form a complete program which is accepted as an input by a typical program model checker (the problem of missing environment). Therefore, a solution is to create an environment of some form for the component that is subject to model checking.

We proposed to model the environment on the basis of a particular component architecture the target component is expected to be used in; the architecture being a context for the component. Specifically, since we aim at hierarchical component architectures with component behavior modeled via behavior protocols, we model the component's environment behavior with an environment protocol computed from frame protocols of other components taking part in the given architecture.

We have presented an algorithm for computing the environment protocol, which is based on syntactical expansion and substitution of frame protocols. Finally, we showed that the solution for modeling the environment's behavior on the basis of the environment protocol is more efficient than our previous approach [9] based on an inverted frame protocol; the reason for this is that the environment protocol reflects the real usage of the target component in the given architecture, while the inverted frame protocol specifies the most general environment for the component.

As for future work, we plan to extend our current algorithm for computing the environment protocol with support for components that call methods on their required interfaces autonomously.

Acknowledgement

We would like to record a special credit to Jan Kofron for valuable comments regarding the design and implementation of the algorithm for computation of the environment protocol.

References

[1] Adamek, J., and F. Plasil, *Erroneous Architecture is a Relative Concept*, Proceedings of Software Engineering and Applications (SEA), ACTA Press, pp. 715-720, Nov 2004

[2] Allen, R., "A Formal Approach to Software Architecture", PhD Thesis, School of Computer Science, Carnegie Mellon University, 1997

[3] Clarke, E. M., O. Grumberg, and D. Peled, "Model Checking", MIT Press, 2000

[4] Clarke, E. M., D. E. Long, and K. L. McMillan, *Compositional Model Checking*, In Proceedings of the 4th Symposium on Logic in Computer Science, June 1989

[5] Cobleigh, J. M., D. Giannakopoulou, and C. S. Pasareanu, *Learning Assumptions for Compositional Verification*, In Proceedings of 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), April 2003

[6] Giannakopoulou, D., C. S. Pasareanu, and H. Barringer, *Assumption Generation for Software Component Verification*, In Proceedings of the 17th IEEE Conference on Automated Software Engineering (ASE), IEEE CS, 2002

[7] Magee, J., and J. Kramer, *Dynamic Structure in Software Architectures*, Proceedings of FSE4, Oct 1996

[8] Medvidovic, N., *ADLs and dynamic architecture changes*, Joint Proceedings SIGSOFT1996 Workshops, ACM Press, Oct 1996

[9] Parizek, P., and F. Plasil, *Specification and Generation of Environment for Model Checking of Software Components*, Accepted for publication in Proceedings of FESCA 2006, ENTCS, Mar 2006

[10] Parizek, P., F. Plasil, and J. Kofron, *Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker*, Accepted for publication in Proceedings of 30th IEEE/NASA Software Engineering Workshop (SEW-30), IEEE CS, Apr 2006

[11] Pasareanu, C. S., M. B. Dwyer, and M. Huth, *Assume-Guarantee Model Checking of Software: A Comparative Case Study*, In Proceedings of the 6th SPIN Workshop, LNCS, **1680**(1999), pp. 168-183, 1999

[12] Plasil, F., and S. Visnovsky, *Behavior Protocols for Software Components*, IEEE Transactions on Software Engineering, **28**(2002)