

Extracting Behavior Specification of Components in Legacy Applications *

Tomáš Poch¹, František Plášil^{1,2}

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

²Academy of Sciences of the Czech Republic
Institute of Computer Science
plasil@cs.cas.cz

Abstract. A challenge of componentizing legacy applications is to extract behavior specification of suggested components. It is desirable to preserve a relation between the original structure of the source code of a component and the extracted specification; in particular, this is important for both user comprehension and for interpretation of results of any further formal verification. Even though the reverse engineering techniques providing behavior specification have already been applied on object oriented software and components, none of them targets the interplay of both the externally and internally triggered activities on the component's provided and required interfaces from a single perspective. This paper targets the problem in the scope of Behavior Protocols and components given as a set of Java classes accompanied with information on component boundaries. To demonstrate viability of the proposed approach, this technique has been partially applied in the JAbstractor tool for the SOFA component model.

Keywords: Reverse engineering, Component behavior specification

1 Introduction

1.1 Why Behavior Modeling of Legacy Components

Component-based software development eases production of complex systems by composition of precisely defined separated blocks of software – components. Since every component clearly states its purpose and assumptions on the environment in terms of provided and required interfaces, it can be developed independently of the

* This work was partially supported by the Grant Agency of the Czech Republic project 201/08/0266.

other components. The isolation of components prevents from unexpected dependencies between unrelated parts of the system.

Component-based development allows reusing a component in different systems and contexts. Moreover, when a component is equipped with additional information like an abstraction of behavior it exhibits and performance characteristics, various kinds of analysis and verification (compliance of components' behavior, performance) can be applied already at design stage to shorten development cycle and lower the costs. The additional information also serves for documentation purposes and test case generation.

Even though the component technology has become used in practice during last decade, there is still a wide range of legacy applications run and maintained, design of which is far from being component-based. Thus, when such application is being modified, developers can not take an advantage of the component paradigm. Reimplementation from scratch using modern component-based methods would be extremely costly. Instead, it pays off to apply reverse engineering techniques to re-design the application in a component-based way.

In this context, to take advantage of formal verification of component behavior, two reverse engineering tasks have to be accomplished: (i) Extracting static structure (architecture) of the application. This means to identify individual components, their provided and required interfaces, and relations among them in the form of binding between interfaces. (ii) Providing the behavior specification for each component identified in the task (i).

This paper aims at the task (ii), assuming the source code of the application is available. Naturally, an automatized way to extract component behavior specification is very desirable. Even though the reverse engineering techniques providing higher level behavior specification were already applied on object oriented software [3][4][6], and components [17], none of them targets the interplay of both the externally and internally triggered activities on provided and required interfaces from a single perspective.

1.2. Goals and Structure of the Paper

In this paper, we assume a behavior specification/formalism (BF) of the power of finite state machine and a component C implemented in an object oriented language (oo language). The component C conforms to an underlying component model which introduces at least the abstractions of provided and required interfaces, each grouping methods, and the execution model of which is based on method calls triggered by both external and internal activities (threads) of components.

Problem statement: The challenge is to find a mapping among the underlying component model abstractions (and their relations) used in C which can be captured both by BF and the oo language representation of C . Based on the mapping, an automatized transformation from the oo language representation of C into behavior specification of C in BF is to be defined. Here, the challenge is to keep a “reasonable” relation between the structure of implementation and the structure of specification (while preserving important aspects of behavior), although the expressive power of BF and oo language significantly differs. Keeping the relation is very important both

for human comprehension and for interpretation of results of a further formal verification. To achieve this important property, certain degree of over-approximation is necessary despite losing some details. Since the components encapsulate implementation details and thus exhibit externally observable behavior of less complexity, there is a good chance that a “reasonable” specification (in terms of size, accuracy) exists. Nevertheless, this assumes a good architecture structuring which really hides the implementation details. If this is not the case, result of the transformation should serve as a feedback to reverse engineering (task (i)) to structure the application in a better way.

Goal: The goal of this paper is to present a technique for extracting the behavior of a component given as a set of Java classes. The extraction is done in an automatized way and the target formalism is Behavior Protocols [1]. This technique has been partially applied in the JAbstractor tool for the SOFA component model [19].

This goal is reflected in the structure of the paper as follows. The formalism of Behavior Protocols and running example is presented in Sect. 2, while Sect. 3 introduces the technique of behavior extraction. The remaining sections are devoted to evaluation and discussion, related work, and a conclusion.

2 Background

2.1 Running Example

The following example will be used in the paper for illustration of the presented ideas. Fig. 1 depicts a fragment of an information system. The `SessionManager` component is intercepting all communication between user interface and application business logic and manages user sessions. When a user wants to log into the system, `UserInterface` asks `SessionManager` to create a new session by invoking the `createSession` method on the `session` interface. Once the new session id is returned, it is used in subsequent requests (`invokeCommand` method on the `session` interface) to identify the session and user. Within the `invokeComand` method, the session id is checked and if valid, the command is passed together with user information to the business logic. If an inactive session is terminated by `SessionManager`, the user interface is notified via the `uiNotify` required interface. The Java implementation of the `SessionManager` and `RndGenerator` components is in Appendix A.

2.2 Behavior Protocols Basic

The formalism of Behavior Protocols (BP) is a high level specification capturing the finite sequences of method calls allowed on the component provided and required interfaces. Having a BP specification (“frame protocol”) available for each

component in a system, a composition operator serves to detect communication errors.

A primitive term in BP represents: accepting a method call `?interface.method{reaction}`, issuing a method call `!interface.method`, and empty action (`NULL`). Expressions use the following operators: '+' alternative, ';' sequence, '*' repetition, '|' parallel operator. More details are in [1][2].

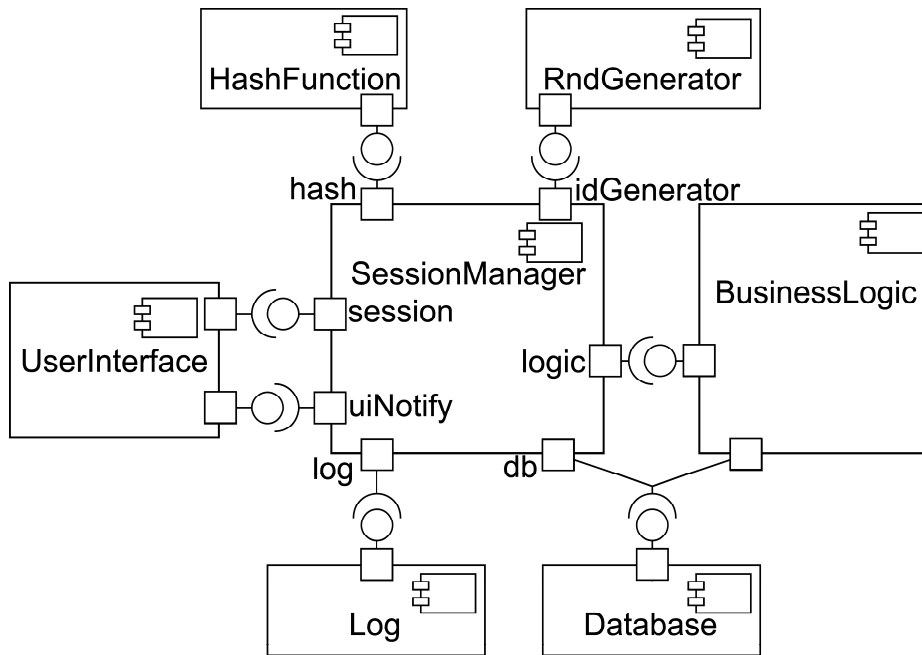


Fig. 1. Fragment of architecture of an information system involving the `SessionManager` component

The BP specification in Fig. 2 captures frame protocol of `SessionManager`. At the beginning, `SessionManager` opens the database connection (`!db.open`). Then, in parallel and repetitively, `SessionManager` can accept calls of `createSession` and `invokeStatement`. Besides, there is an autonomous thread running in parallel, which notifies about session timeout. The reaction on `createSession` first invokes `getHash` on the `hash` required interface to handle the password passed as a parameter. (Since parameters are not considered, just the method call is captured and no relation between parameters is apparent from the specification.) When the hash is returned, it is approved either by querying the database (`!db.query`) or (+ operator) the internal cache of `SessionManager` which does not have any externally observable effect (`NULL`). Depending on the result, new session id is generated (logged in any case).

```

!db.open; (
  (?session.createSession{
    !hash.getHash; (!db.query+NULL) ;
    ((!idGenerator.generate;!log.log) + (!log.log))
  })*
  |
  (?session.invokeStatement{
    !log.log;!logic.invokeStatement+NULL
  })*
  |
  (!uiNotify.sessionTerminated;!log.log)*
)

```

Fig. 2. Frame protocol of the SessinManager component in BP

At semantic level, a method call $i.m$ is represented by pair of events $(i.m\uparrow, i.m\downarrow)$ corresponding to issuing of method call and return from the method call. A component reaction may occur between them. BP expression semantics is given by an LTS labeled with those events. Composition of BP is then defined as synchronous product of the corresponding LTSs.

3 Strategy of Extraction

In principle, extracting a frame protocol from Java code of a component C boils down to finding a mapping among the underlying component model abstractions (and their relations) used in C which can be captured both by the frame protocol and the Java representation of C . Basically, the frame protocol determines what the reaction on a method call on a provides interface should be in terms of activities on the requires interfaces of C (reaction of an external thread activity), and, moreover, can determine autonomous activities on the required interfaces driven by an internal thread of C . Finally, the frame protocol determines what the corresponding interplay of both the externally and internally triggered activities should be (potential sequencing/interleaving, alternative execution, and repetition). Thus all these features have to be identified in the Java representation of C and mapped to the BP representation of the frame protocol of C . The following subsections describe how this can be done in three steps: (i) Extraction of reactions on method calls, (ii) Identification of autonomous activities (threads active on required interfaces, initialization), (iii) integration step defining the desired interplay of all activities (both external and internal).

In description of these steps, the Java representation of C is assumed to be provided in the form of a set of Java classes and architectural information mapping the Java concepts to the abstractions of the target component model. The architectural information consists of five Java declaration sets: $C^{Classes}$, $C^{Required}$, $C^{Provided}$ and C^{Init} . Here, $C^{Classes}$ contains the Java classes instantiated by the component model runtime infrastructure when a C 's instance is being created. Then, $C^{Classes^*}$ contains all classes

instantiated (transitively) from an element of C^{Classes} . C^{Required} contains the member variables referencing the C 's required interfaces, while C^{Provided} contains the Java interfaces implemented by classes from C^{Classes} (these Java interfaces correspond to C 's provided interfaces). Finally, C^{Init} contains the initialization methods called by the component model runtime infrastructure when C is instantiated. The following properties hold for these sets (the relations here are defined intuitively): $A \in C^{\text{Classes}} \Rightarrow A \in C^{\text{Classes}}$; $A \in C^{\text{Classes}} \wedge A \text{ instantiates } B \Rightarrow B \in C^{\text{Classes}}$; $A \in C^{\text{Required}} \Rightarrow A.\text{parent} \in C^{\text{Classes}}$; $A \in C^{\text{Provided}} \Rightarrow \exists B \in C^{\text{Classes}}$ implementing A ; $A \in C^{\text{Init}} \Rightarrow A.\text{parent} \in C^{\text{Classes}}$.

3.1 Extraction of a Reaction on a Method Call

The first step aims at capturing the BP specification of the effect of each method m from a C 's provided interface. Precisely, m is a method of a class from C^{Classes} and implements an interface from C^{Provided} . The effect of the method m is captured in terms of the corresponding events on the required interfaces of C – the specification captures the (externally) observable behavior performed by the component in reaction on any call of m . The basic idea is to “inline” all the internal objects' methods which appear in the calling chains defined by the Java body of m . The result of inlining is a method m^{inl} which contains only method calls on interfaces determined by C^{Required} .

For illustration, consider the implementation of the `createSession` method provided by `SessionManager` (Appendix A, line 37). Obviously `createSession` first invokes `getHash` on the `hash` interface. Then, the login name and password are checked and if valid, new session identifier is generated and returned as result. The validation is implemented by the internal object `DBCACHE`, which is accessing the required interface `db`.

In principle, the extraction omits all the behavior of m inexpressible in BP. The method m^{inl} is created from m in the following steps. First, all variables, parameters, and their usages are omitted, as well as the method calls that do not lead to a call on a required interface from C^{Required} . Then, method inlining is recursively applied, so that in m^{inl} remain only the method calls on members of C^{Required} linked by control flow statements. Finally, empty control flow statements are removed. To convert the resulting m^{inl} into BP notation, the effect of Java control statements is transformed as defined in Table 1. As apparent from the table, all conditions are replaced by non-determinism (an over-approximation).

For example, the resulting BP specification of the `createSession` method takes the form

```
createSession {!
    hash.getHash; (!db.query+NULL) ;
    ((!idGenerator.generate;!log.log) + (!log.log))
}
```

Table 1. Java statement to BP transformation

Java statement S	Corresponding protocol P(S)
<code>requiredIf.method(p1, . . . , pn)</code>	<code>!requiredIf.method</code>
<code>S1;S2</code>	$P(S_1);P(S_2)$
<code>for(i;c;it) S1</code>	$P(S_1)^*$
<code>while(c) S1</code>	$P(S_1)^*$
<code>if (c) S1 else S2</code>	$P(S_1) + P(S_2)$
<pre>switch (e) { case a: Sa case b: Sb ... default Sd }</pre>	$P(S_a)+P(S_b)+\dots+P(S_d)$
<i>empty statement</i>	NULL

There are more challenging issues in forming the reaction on a method call: (i) Any method m_i in Java is a virtual method, so that the actual implementation of m_i is selected at runtime. Thus, when a method m_i is being inlined into m , the call of m_i has to be replaced instead of single method body by a non-deterministic choice (the +operator) from all of the available implementations of m_i . In principle, points-to analysis [10] can be used to narrow the set of m_i implementations to those that can be actually called. (ii) All the method calls on the required interfaces of C have to be identified in the Java implementation. Although the fields referencing a required interface are statically captured in C^{Requires} , the reference can be copied to another variable (duplicated).

For illustration, the `DBCACHE` class (Appendix A, line 16) keeps the reference on required interface in variable `db`. This variable is initialized in the constructor, however it is not annotated. In this respect, a naïve solution is to consider all the variables sharing their type with a variable from C^{Requires} also as referencing a required interface. However, this would not allow distinguishing two interface instances of the same type. A better solution is to use points-to analysis to distinguish different reference targets. (iii) During inlining, a recursive method may occur. Such recursive method definitely involves also a method call on an interface from C^{Required} (otherwise it would be omitted in the previous step). Since exact number of iterations is not captured in BP anyway, such recursion can be replaced by non-deterministic loop.

Finally, the resulting BP specifications of m^{inl} have to be combined together – how this is done is described in Sect. 3.3

3.2 Extraction of Autonomous Activities

The goal of the second step is to create specification of the behavior performed by C on its own (as an active entity). In terms of Java, this means to analyze the behavior performed by the C 's internal threads and initialization methods (an initialization

method appears as an activity of C , since it is called by the underlying run time infrastructure of the component model, not by another component). Since this behavior is also explicitly expressed in Java, similar technique as the one described in Sect. 3.1 can be used for its extraction (transformation into BP). Initialization methods are given by the architectural information in C^{init} .

A thread is considered to belong to C if its class is in C^{Classes} and it is started by a method of a class in C^{Classes} , and it is accessing C^{Classes} . However, a thread belonging to C that does not access any required interface does not contribute to the observable behavior of C and thus it is not interesting for the extraction.

For illustration, `cleanupThread` (Appendix A, line 79) is created and started during the initialization of `SessionManager`; it invokes `cleanupOldSessions` which modifies the data structure keeping the set of current sessions. Thus, the thread belongs to the `SessionManager` component. The result of this step is the BP specification of `init` and `cleanupThread`. Since `init` does not feature any (externally) observable behavior, its BP specification is `db.open`. The BP specification of `cleanupThread` can take the form `(!uiNotify.sessionTerminated ; log.log)*`.

The thread concept in Java is quite powerful. There is no restriction posed on the number of thread instances, on the moment of start and termination, etc. On the other hand, due to the static nature of BP specification, extraction of thread behavior is possible only in specific cases though, e.g. for threads started within the initialization of C . This issue is more discussed in Sect. 5.

How the BP specifications produced in this step are combined together and how they are integrated with the results of the method reaction step is described in Sect. 3.3.

3.3 Integration Step

Finally, the third step puts the results of the previous steps together to state how the component C should be called in its provided interfaces and what the corresponding interplay of calls on its required interfaces is. As mentioned in Sect. 2.2 such specification of C 's behavior expressed in BP is the frame protocol of C . The frame protocol states restrictions on the C 's environment in terms of the allowed sequences/interleaving of method calls and internal thread actions. In contrast to the previous steps, information about the important mutual dependencies between individual code fragments is present in the primitive component code only in a "scattered way", since each method can contribute to the C 's state modification differently and each state history implies specific desired call ordering. Moreover, the resulting frame protocol may also depend upon the actual usage of the component. However, the integration of the results of the previous two steps into a frame protocol is not a trivial task, having a straightforward generic solution. The following three approaches to frame protocol integration we found after many experiments most viable:

(a) *Maximal parallelism preventing race conditions*. If the C 's implementation is protected from race conditions by an appropriate Java synchronization both in the provided methods and internal threads (the synchronization can be statically checked

in both cases), the methods can be called in any order, even in parallel, and also in parallel with all the internal threads. Assuming C is initialized before any other action, the frame protocol takes the form

$$\text{init}; ?m_1\{r_1\}^* | ?m_2\{r_2\}^* | \dots | ?m_n\{r_n\}^* | t_1 | \dots | t_m \quad (\text{I})$$

where init is BP capturing the initialization phase, m_i is a name of a method, r_i is the reaction of m_i , and t_i is BP is specification of a thread. An example of (I) is the protocol in Fig. 2. When C is designed as single threaded (parallel execution of methods as well as existence of individual threads is not considered) the frame protocol may take a simpler form

$$\text{init}; (?m_1\{r_1\} + ?m_2\{r_2\} + \dots + ?m_n\{r_n\})^* \quad (\text{II})$$

The frame protocols (I) and (II) are special cases considering all methods to be executed in parallel resp. sequentially. In general, however, C may feature groups of methods not allowed to be executed in parallel. Those groups may be identified by static analysis. In particular, static analysis can identify the pairs of methods that alternatively (i) may cause a race condition (may access the same variable without locking), (ii) are mutually excluded when accessing a shared variable, (iii) do not share a variable. To precisely characterize the pairs, we define a symmetric relation $\text{RC}(m,n)$ to hold for methods m and n if parallel execution of those methods may lead to race condition. Then, RC^\bullet is a reflexive transitive closure of RC . RC^\bullet is an equivalence relation and divides the set of methods into equivalence classes. Then, for each equivalence class E_i we create a protocol P_{E_i} by connecting methods call acceptance by $+$ operator. Using these equivalence classes, the integrated frame protocol takes the form:

$$\text{init}; P_{E_1}^* | \dots | P_{E_k}^* | t_1 | \dots | t_m \quad (\text{III})$$

Such protocol, however, does not allow executing a method m many times in parallel even though the parallel execution can not cause a race condition ($\neg\text{RC}(m,m)$). To reflect this fact in the frame protocol and to allow g parallel executions of m , all occurrences of $?m$ in (III) can be safely replaced with $?m | \dots | ?m$, where m is repeated g times. The constant g can be determined from the component's environment [16].

As an example, consider again the `SessionManager` component. It provides two methods – `createSession` and `invokeCommand`. Although these methods share the field `activeSessions`, access to it is guarded by Java synchronization. Thus, these methods belong to different equivalence classes and the frame protocol takes form presented in Fig. 2 in Section 3.2. On the other hand, when the `RndGenerator` implementation from Appendix A is considered, there are also two methods – `generate` and `seed`, however in this case both of them access the same variable and the access is not mutually excluded. Thus, $\text{RC}(\text{generate}, \text{seed})$ holds and consequently the frame protocol takes the form $(?randomGenerator.seed + ?randomGenerator.generate)^*$.

Note, that any pair of synchronized methods m_i and m_j (satisfying (ii)) is handled as if they were totally independent and satisfying (iii), falling thus into different equivalence classes. This is an over-approximation, since synchronized m_i and m_j

bodies are never executed in parallel, even though the calls of m_i and m_j can be accepted in parallel.

(b) *Preventing the component from reaching an invalid state.* Although the frame protocols (I) – (III) indicate how an environment’s usage should avoid race conditions within the component implementation, they still do not capture design dependencies of individual methods in terms of the desired sequences of their usage. As an example, consider the `RndGenerator` component which assumes the `seed` method is called before using the `generate` method (assume also, that this is checked in the code via an assertion – Appendix A, line 93). Thus, the resulting frame protocol of `RndGenerator` should take the form `(?idGenerator.seed; (?idGenerator.generate)*)*` to express this restriction and, as an aside, also the fact that `RandomGenerator` does not have to be used at all.

In general, if such a restriction is violated, the component is to reach an error state. To avoid the violation in the extracted frame protocol, such error states have to be explicitly determined. This can be done either directly by the user (e.g. by checking a predicate over member variables), or even implicitly (e.g., no low level exception is thrown). When error states are determined, elaborated approaches using model-checking can be used [4][6] to obtain the desired call sequencing. Here, the result of these methods is typically an LTS capturing the allowed sequences of method calls (parallelism is not considered). To form a frame protocol containing just method acceptance in BP, the LTS can be easily transformed to Behavior Protocol P using the algorithm from [12] since BP is based on regular expression. Then, the frame protocol is finalized by enhancing R by the reactions (such as r_i in (I)) provided by the Step 1 (Sect. 3.1).

(c) *Analysis of a complete application.* So far, approaches considering an isolated component C were discussed. However, having available an application using C correctly (in another words, an environment E_C of C is available), a valid usage specification can be obtained by analyzing E_C . Obviously, E_C must be correct in terms of C’s usage and the result of the analysis does not yield the most general frame protocol of C, but captures only the behavior exhibited and employed by E_C . This can be improved by analyzing more applications using C (environments of C) and integrating the results. There are basically three approaches for analyzing E_C – runtime monitoring [3], static analysis [5] and state space traversal [13]. None of them, however, considers parallelism.

Overall, in order to provide a frame protocol featuring maximal parallelism, while still keeping the necessary sequencing of methods and preventing race conditions, the result provided by the approaches (b) and/or (c) can be improved by adding information from (a). Assuming the result of (b) or (c) in the form of BP, parallelism can be introduced in certain subexpressions of the repetition operator. Let us suppose a Behavior Protocol P in form $(P_1 + P_2 + \dots + P_n)^*$. As an aside, in a special case, P may thus take the form P_1^* . Then, we define a relation RC_P over the set of alternatives $\{P_1, \dots, P_n\}$. The relation RC_P holds for alternatives P_i and P_j if their parallel execution may lead to a race condition. RC_P is defined using the relation RC since their motivation is similar. In particular, $RC_P(P_a, P_b)$ holds if there is no method p invoked in P_a and no method q invoked in P_b such that $RC(p,q)$. Similar to the relation RC, the relation RC_P is symmetric and its reflexive transitive closure is denoted as RC_P^\bullet . Then, for each equivalence class E_i of RC_P^\bullet we define a protocol P_{E_i} by

connecting the alternatives from the equivalence class by the + operator. Then, the original protocol P can be replaced by protocol

$$P_{E1}^* | P_{E2}^* | \dots | P_{Ek}^* \quad (\text{IV})$$

Moreover, certain alternatives can run many times in parallel without risking a race condition which should be reflected in the frame protocol (alternatives P_i such that $\neg RC_P(P_i, P_j)$). Then, all occurrences of P_i in (IV) can be replaced by g copies of P_i connected by the parallel operator to allow g executions of P_i in parallel.

For illustration, consider the result obtained by method (b) for the implementation of the `RndGenerator` which takes form `(?idGenerator.seed; (?idGenerator.generate)*)*`. There are two repetition operators. There are no alternatives in the top level repetition operator (P_1^* form), so there is just one equivalence class E containing single protocol P taking the form `?idGenerator.seed; (?idGenerator.generate)*` and the associated protocol P_E is the same. Since $RC(\text{seed}, \text{generate})$ holds, the relation $RC_P(P_E, P_E)$ holds too. Thus, we can not put more protocols P_E in parallel in the result. In contrary, when the second repetition operator is taken in account, there are also no alternatives, thus there is single equivalence class and the associated protocol P_E takes the form `?idGenerator.generate`. However, here $RC_P(P_E, P_E)$ does not hold, as $\neg RC(\text{generate}, \text{generate})$ and there is no other method in P_E . Thus, the overall result allowing two parallel invocations of the `generate` method takes the form `(?idGenerator.seed; (?idGenerator.generate)* | ?idGenerator.generate)*`.

4 Prototype Tool: JAbstractor

To evaluate and verify viability of the ideas presented in the previous sections, a proof-of-the-concept prototype tool (JAbstractor) has been developed. As input, JAbstractor takes Java source code of a primitive component. The code has to be compilable, which implies that also declarations of all the employed types are to be available. Since the static architecture reconstruction is separate task the architectural information (sets from Sect. 3) is defined via Java annotations directly in the code (those annotations can be obtained by an third party tool [7] or specified manually).

The current JAbstractor version extracts reactions on method calls (Sect. 3.1) using a sequence of AST transformations. It also detects autonomous threads created and started within initialization methods and extracts their behavior. Regarding the integration step of behavior extraction, just the simple solution using the frame protocol in the form (I) presented in Sect. 3.3 has been implemented so far.

To create an AST of a source code, JAbstractor uses the Recoder tool [9], which provides also an AST transformation framework. This is advantageously employed for the transformation of the provided methods (m to m^{int} , Sect. 3.1) and autonomous activities (Sect. 3.2) – they are performed on their AST representations. Then, for each class A_i from $C^{Classes}$ a new Java class, $Merged^{A_i}$, is created. It represents the component's behavior exhibited through each interface I from $C^{Provided}$ implemented

Each $\text{Merged}^{\text{Ai}}$ class contains also initializing methods and fields for threads obtained in Sect. 3.2. In the source AST, threads are identified by invocation of the start method on the Thread class. The $\text{Merged}^{\text{Ai}}$ classes do not instantiate any other classes than threads, and do not contain other fields than those corresponding to C^{Required} . The final AST transformation translates the ASTs of $\text{Merged}^{\text{Ai}}$ classes into a BP AST to integrate the frame protocol in the form II introduced in Sect. 3.3.

The design of the tool is modular. In Sect. 3.1, it is suggested to use points-to analysis to improve results. Since we do not plan to provide an implementation of points-to analysis on our own, the tool just provides an interface to a general points-to analysis. Currently, there is just a naïve implementation based on type information; however, we plan to provide the points-to analysis implemented in the SOOT framework [11] through the general interface. Another modular aspect of JAbstractor is the way the architectural information is specified. There is an interface providing the information in the form of sets (Sect. 3) to the rest of the tool. The current implementation obtains the architectural information from Java annotations capturing the relation of declarations in the code to the abstractions of the SOFA component model.

5 Discussion and Related Work

In principle, an execution of a Java program on a specific computer can be modeled by FSM. The number of states, however, is extremely large bounded by $2^{\text{memory_size}}$. In any case, such a FSM, besides being beyond the abilities of human comprehension, does not reflect the original structure of the Java program.

Nevertheless BP has the power of FSM. To make a BP specification useful, keeping the relation between the original structure of the component Java implementation and the extracted BP specification is very important both for human comprehension and for interpretation of results of a formal verification. To achieve this important property, certain degree of over-approximation via non-determinism is necessary despite losing some details. In following paragraphs we focus on the effect of over-approximation employed in the extraction steps from Sect. 3.

Extraction of a reaction on a method call (Step1). A consequence of ignoring data and especially conditions in the extraction method proposed in Sect. 3.1 and Sect. 3.2 is the introduction of non-determinism into each control flow statement (over-approximation). Specifically, a loop with a number n of iterations is replaced by a finite number of iterations as well as bounded recursion. Similar comment goes to branching statements.

On the other hand, since the inlining is a straight-forward process, it always produces a result. Thus, there are no limits on the complexity of the interplay of activities on component's interfaces, however if the communication is really complex, the over-approximation can lead to spurious errors in a further formal verification (such as detection of communication errors as presented [14]).

Extraction of autonomous activities (Step 2). It has to deal with threads, their start and termination and also synchronization. Start is simple, but termination and synchronization relies on component state and method parameters (“data”), which are, however, ignored by inlining. Moreover, threads modeled in BP via parallel operator are limited to those which join in the context of the BP operator enclosing the parallel operator. For example in the protocol $?m\{t_1\ t_2\}$ (i.e., $?m\uparrow; t_1\ t_2;!m\downarrow$), neither t_1 nor t_2 can survive return from the method m . In consequence, a thread started in a method m and running beyond the return of m , can be modeled in BP only by dividing the thread into two sections – running inside m and another outside m . Similar issue is discussed in [15]. Also in this case, to reflect all possible divisions, the complexity of BP specification grows so that direct relation to the Java thread is blurred. Nevertheless, without considering data, it is hard to identify termination of Java threads. Thus, currently just the threads started during initialization and running indefinitely are supported.

Integration step (Step 3). The technique (a) inherently involves over-approximation. On the other hand, as mentioned in the conclusion of Sect. 3.3, combining the techniques (a) and (b) reduce the level of over-approximation. Nevertheless, the abstraction achieved by forming the frame protocol in the form (IV) is an important advantage in terms of explicitly specifying the required sequencing of methods calls when using the component. This is very important for an easy comprehension of the correct component usage, since this information is otherwise not apparent from Java interfaces (is hidden in the method code).

Overall. From the discussion above, it follows a need to reflect explicitly data in the extracted specification to lower the level of over-approximation while preserving a close relation to the Java code structure. There is already a candidate to replace BP extraction result form: the TBP specification [18] supports both, explicit notion of data (just enumeration types) and threads closer to Java capabilities (still a specific number).

Although introduction of data in the extraction process is a subject of future work, we have already identified individual steps and issues to be solved. First, it is necessary to identify data important for component “business” behavior. This typically means to identify member variables keeping the state of the component, method parameters significantly influencing the control flow of a method. Then, since TBP provides enumeration types only, a mapping of a source variable type into an enumeration type or replacement of a source variable by several variables of enumeration types have to be provided (introduces an abstraction). Identification of the important data and necessary mappings can be done manually by the user having a deep insight into the application structure. On the other hand, an automatized way seems to be possible: Quite promising is the idea of predicate abstraction.

When the information about important data is available it must be reflected by the extraction steps described in Sect. 3. In particular, inlining of methods implemented by different objects must also consider merging of those object’s data. It is also desirable to employ again points-to analysis to identify the variables referencing the

same instance. Also, statements and conditions referencing these important variables may not be considered as dead code.

Although whole paper considers extraction of a component's frame protocol in BP from the Java implementation, the overall idea, as well as individual steps, can be applied to different formalism of similar expressive power (FSM, resp. finite LTS) provided a suitable mapping of component model abstractions (provided/required interfaces, method call, autonomous activity) into the formalism is available.

Related work. A typical use case of a behavior specification extraction method is to make it easier to implement changes in legacy applications by describing desired interplay of calls on components interfaces and their relation to component autonomous activities. However, to enable the extraction and further formal verification, the architecture of a legacy application must also be provided. Here, the work [8] compares different approaches to static architecture extraction on general level. More specifically, the method [7] applies certain heuristics by means of static analysis of Java code to identify components and relations among them.

While the architecture extraction is a prerequisite to our method when dealing with legacy application, the following works aim at similar goal, even in a reduced form: They focus only on specification of method call acceptance (not considering required interfaces). None of them, however, deals with software components at such detail as we do (instead they deal with artifacts like classes and objects). Here, the approaches can be divided to several groups according to the technique used. The most straightforward technique is to employ monitoring of a running program to collect the information and then provide the specification capturing the monitored behavior. This is discussed in [3] where machine learning and stochastic methods are used to extract the behavior specification. Moreover, when the source code of the whole application is available, static analysis techniques can be used instead of monitoring. Quite advanced technique belonging to this other group is described in [5]. In particular, abstract interpretation is used to obtain abstract traces which are further transformed into a finite automaton. Obviously, this technique assumes correctness of the application. Another group form static analysis methods used on isolated classes; a representative can be found in [6], where, first, Boolean abstraction of the class is created using predicates provided by the user. The resulting abstraction is translated into the form required by the SMV model checker. Then, the model checker is queried by the L* algorithm to learn a finite machine model of the most general environment which does not allow the object to enter an invalid state.

Extraction of method effects from component implementation in Java via static analysis is presented in [17]. The technique is similar to our technique extracting method call reaction described in Sect 3.1. In addition, they focus on apparent relations between method parameters and branches and loops in the extracted method specification.

6 Conclusion and Future Work

In this paper we present a technique for an automated transformation of Java representation of SW components to their behavior specification in BP formalism. The technique is based on extraction of method reactions and extraction of autonomous activities. Results are integrated with the aim to maximize parallelism, to prevent the component to enter an invalid state by expressing the necessary call ordering. The technique was partially applied in the JAbstractor prototype tool.

In addition to propositions for future research mentioned in Sect. 5 (in particular replacing BP by TBP), more work has to be done on the JAbstractor tool. Currently, we are about to replace the Java representation provided by the Recoder tool by an EMF model. A long term goal is to develop a framework for evaluation of various extraction techniques able to exploit third party static analysis and model-checking tools.

References

- [1] Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002
- [2] Kofron, J.: Behavior Protocols Extensions, Ph.D. thesis, Charles University, Sep 2007, <http://dsrg.mff.cuni.cz/~kofron/phd-thesis/>
- [3] Ammons, G., Bodík, R., Larus, J. R.: Mining specifications, Proc. of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2002, Portland, Oregon
- [4] Whaley, J., Martin, M. C., Lam, M. S.: Automatic extraction of object-oriented component interfaces, Proc. of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, 2002, Roma, Italy
- [5] Shoham, S., Yahav, E., Fink, S., Pistoia M: Static specification mining using automata-based abstractions, Proc. of the 2007 Int. Symposium on SW testing and analysis, 2007, London, United Kingdom
- [6] Alur, R., Černý, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java Classes. Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2005.
- [7] Chouambe, L. Klatt, B. Krogmann, K.: Reverse Engineering Software-Models of Component-Based Systems, Proc. of CSMR 2008
- [8] Bowman, I. T., Godfrey, M. W., Holt, R. C.: Extracting source models from Java programs: Parse, disassemble, or profile? ACM SIGPLAN Workshop on Program Analysis for SW Tools and Engineering Toulouse, France, 1999.
- [9] Recoder Tool: <http://apps.sourceforge.net/mediawiki/recoder/>
- [10] Schwartzbach, M. I.: Lecture Notes on Static Analysis, 2008
- [11] Lhoták, O., Hendren, L.: Scaling Java Points-To Analysis using Spark, LNCS Volume 2622/2003
- [12] Hopcroft, J. E., Motwani, R. Ullman, J. D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 2007

- [13] Giannakopoulou, D., Păsăreanu, C., Interface Generation and Compositional Verification in JavaPathfinder, Fundamental Approaches to Software Engineering 2009, LNCS 5503, 2009
- [14] Jezek, P., Kofron, J., Plasil, F.: Model Checking of Component Behavior Specification: A Real Life Experience, In Electronic Notes in Theoretical Computer Science, Vol. 160, Elsevier B.V., Aug 2006
- [15] Parizek, P., Plasil, F.: Modeling of Component Environment in Presence of Callbacks and Autonomous Activities, In Proc. of TOOLS EUROPE 2008, Springer-Verlag, LNBIP, vol. 11, Jun 2008
- [16] Adamek, J.: Verification of Software Components: Addressing Unbounded Parallelism, Int. J. of Computer and Information Sci., Vol. 8, Num. 2, 2007
- [17] Kappler, T., Koziolok, H., Krogmann, K., Reussner, R: Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. Proc. of Software Engineering, 2008
- [18] Kofron, J., Poch, T., Sery, O.: TBP: Code-Oriented Component Behavior Specification, Accepted for publication in Proc. of SEW-32, IEEE, 2009
- [19] Bures, T., Hnetyka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proc. of SERA 2006, IEEE CS

Appendix A

Implementation of `SessionManager` and `RndGenerator` in Java. Annotations provide mapping of Java declarations into component model constructs.

```

1 @Provided interface Session {...}
2 @Provided interface RndGenerator {...}
3
4 public class SessionManager implements Session {
5     /* required interfaces */
6     @Required(name = "logic") BusinessLogic logic;
7     @Required(name = "hash") Hash hash;
8     @Required(name = "log") Log log;
9     @Required(name = "idGenerator")
10         RndGenerator idGenerator;
11     @Required(name = "uiNotify")
12         GUINotification uiNotify;
13     @Required(name = "db") Database db;
14
15     Map<Long, Date> activeSessions = new HashMap();
16     private DBCache dbCache;
17
18     private static class DBCache {
19         Map<String, String> userCache= new HashMap();
20         Database db;
21         public DBCache(Database database){
22             db = database;

```

```
23     }
24
25     public boolean loginQuery(String login,
26         String hashedPasswd){
27         String oldHashedPwd = userCache.get(login);
28         if (oldHashedPasswd==null){
29             if (db.loginQuery(login, hashedPasswd)){
30                 userCache.put(login, hashedPasswd);
31                 return true;
32             }
33             return false;
34         } else {
35             return hashedPasswd.equals(oldHashedPwd);
36         }
37     }
38 }
39
40 private void touchSession(long sessionId){
41     activeSessions.put(sessionId, new Date());
42 }
43
44 public synchronized long createSession(
45     String login,String passwd) {
46     String hashedPasswd = hash.getHash(passwd);
47     if (dbCache.loginQuery(login, hashedPasswd)){
48         long sessionId = idGenerator.generate();
49         log.log("User "+ login +" access granted");
50         touchSession(sessionId);
51         return sessionId;
52     } else {
53         log.log("User "+ login +" access denied");
54         return 0;
55     }
56 }
57
58 public synchronized String invokeCmd(
59     long sessionId, String command) {
60     String ret = null;
61     if (activeSessions.containsKey(sessionId)){
62         ret = logic.invokeCommand(command);
63         log.log("Command accepted: "+command);
64         touchSession(sessionId);
65     } else log.log("Command rejected: "+command);
66     return ret;
67 }
68
69 private Date getOldestTS(){...}
70
```

```
71  private void cleanupOldSessions(){
72      Date oldestTimestamp = getOldestTS();
73      Iterator it =
74          activeSessions.entrySet().iterator();
75      while (it.hasNext()) {
76          Entry<Long,Date> e = it.next();
77          Date entryDate = e.getValue();
78          if (oldestTimestamp.before(entryDate)){
79              it.remove();
80              uiNotify.sessionInvalidated(e.getKey());
81          }
82      }
83  }
84
85  @Start public void init(){
86      db.open("dbLogin", "dbLogin");
87      idGenerator.seed(1);//derived from time
88      dbCache = new DBCache(db);
89      Thread cleanupThread = new Thread(){
90          public synchronized void run() {
91              while (true) {wait(30000);}
92              cleanupOldSessions();
93          }
94      };
95      cleanupThread.start();
96  }
97
98 }
99
100 public class RandomGeneratorImpl
101     implements RndGenerator
102 {
103     long seed = -1;
104     public long generate() {
105         assert (seed!=-1);
106         return seed%13;//TODO: generate better
107     }
108     public void seed(long s) {
109         seed = s;
110     }
111 }
```