Runtime Validation of Communication Histories

Einar Broch Johnsen, Gerardo Schneider, and Øystein Torget
Department of informatics, University of Oslo
PO Box 1080 Blindern, N-0316 Oslo, Norway
{einarj, gerardo, oysteto}@ifi.uio.no

Abstract

Component based software development techniques are becoming increasingly popular, as they improve the software development process through component reuse. However component based development poses a challenge to software verification: How can we assert the correctness of a black-box component without having access to the internal logic of its implementation? In this paper, we propose an approach to this challenge by validating a component's communication history with respect to a specification of its observable behaviour using runtime verification techniques. For this purpose we present a simple specification language for describing component behaviour in terms of communication protocols, a language extension to support error handling at the communication level, and a prototype tool to monitor components and assert that they satisfy their protocol specification at runtime. The prototype is implemented for Java components, supports multithreaded access to the monitored components, and is demonstrated on two examples.

1 Introduction

In order to construct software systems by component composition, the components must communicate correctly with each other to successfully make use of the services provided by other components. Correct communication in this sense is not merely restricted to respecting type constraints on data values, but will typically consist of a notion of component *protocol* describing a component's provided services in terms of the meaningful sequences of interaction in which the component can participate. In general it is not easy to ensure that these protocols are not violated, and multithreading makes this task even more difficult. In contrast to testing and simulation, formal verification techniques like theorem proving and model checking guarantee correctness by exploring all possible executions. These techniques have been used successfully in both academic and industrial applications, but their applicability remains restricted to specific kinds of systems.

Runtime verification is an interesting and complementary technique for program validation. Specifications are made in a formal language and the program is monitored for events that are relevant to this specification. The trace of events is analysed on-the-fly with respect to the specification. Since only a single trace is analysed, runtime verification is expected to scale well. Several frameworks for runtime verification exist, including Java PathExplorer (JPAX) [10] and Java-MOP [3] for Java programs. JPAX supports specifications in temporal logic and features algorithms for concurrency analysis which identify race conditions and deadlocks. Java-MOP emphasises monitoring as an important part of software design; the framework is specification independent and supports different specification languages through *logic plug-ins*.

An important application area for runtime verification tools, such as JPAX and Java-MOP, is the validation of communication histories. The following example motivates this need. The application depicted in Fig. 1 consists of a web browser, an application front-end, a user manager, a query manager, and a database. The web browser accepts user input and provides the user interface. The application front-end handles requests from the web browser and directs them to a component that can handled the requests. The user manager authenticates users, gives each authenticated user a session identifier, and keeps track of all active session identifiers. The query manager handles queries to the database which contains all relevant data for the application. In this system there can be several constraints on what is considered valid communication between the components. Three examples of such constraints are (1) the user manager should not communicate with the query manager, (2) the query manager should only send valid queries to the database, and (3) the user manager should not provide an active user with a new session identifier if the user authenticates himself again, while he is still active.

In object-oriented languages like Java, communication between components is mostly through method calls. Two

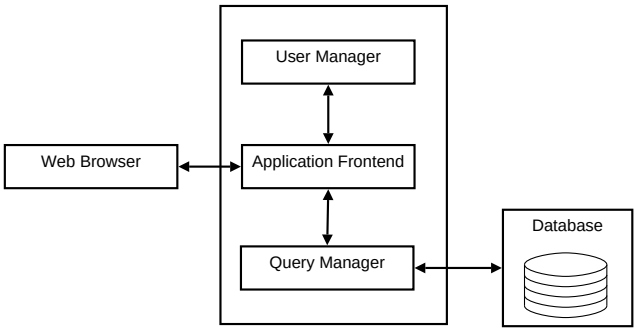

Figure 1. An example web application.

ways of specifying restrictions on method calls are *access modifiers* and *type restrictions* [7]. Access modifiers restrict method access to certain components and type restrictions ensure that the arguments to method calls are within certain boundaries. Both access modifiers and type restrictions capture restrictions related to one method call. They do not take the history of communication into account and do not place restrictions on the communication protocol. In the example above restrictions 1 and 2 can be captured by access modifiers and type restrictions, respectively. However, restriction 3 cannot be specified without taking the history of communication into account and cannot be captured by access modifiers or type restrictions.

This paper presents a runtime verification approach that focuses on the specification and validation of *communication histories*. No knowledge of the internal logic of a component is required; the approach only considers its external behaviour, i.e., its communication with other components. For this purpose, we introduce SSL, a simple specification language for specifying communication protocols, based on regular expressions.

Paper overview. The next section gives an overview of the approach. In Sec. 3 we present our specification language while in Sec. 4 we briefly describe the validation algorithm. We show how to treat runtime errors in Sec. 5. Sec. 6 presents the prototype and Sect. 7 show some examples and benchmarks. We then discuss related and future work and conclude the paper.

2 Overview of the Approach

Our approach views a software system as a set of components that communicate with each other through some form of *message passing*. By *component*, we understand a part of a software system which offers predefined services to other parts of a system and which is able to communicate with these other parts. A component can itself be built from smaller components and be part of a bigger component. The part of the software system that is not part of the considered component is called the component's *environment*.

All communication between the considered component and its environment from the moment the component was created up to some time t, is called the component's *communication history* up to t. We represent the communication history of the component as a *message sequence*. For both finite and infinite executions of the component, the communication history of the component up to time *t* is always finite. Therefore the message sequence is finite as well. We assume that a *message* is the smallest unit of communication that can occur between a component and its environment.

We divide the validation engine into three main parts: *Specification*, *Monitor*, and *Validator*. The Specification is a representation of the communication protocol between a component and its environment. The protocol expresses what is considered a correct message sequence for the component's services. Protocols are represented using *extended finite state machines* (EFSM). A message sequence that is correct with respect to the specification, *satisfies* the specification; otherwise we say, it *violates* the specification. The Monitor is responsible for recording communication between components, which is passed as a message sequence to the Validator. The Validator is responsible for validating that the provided message sequences satisfy the specification. The relationship between the different parts is shown in Fig. 2. A message is processed in the following order (the numbers in the list corresponds to the numbers in the figure):

1. The environment sends a message to the component. The message is intercepted by the Monitor.
2. The Monitor forwards the intercepted message to the Validator.
3. The Validator checks that the message is correct with respect to the specification. If the message is not correct the Validator flags the message as an error to be handled by the system.
4. The message is sent to the component.
5. The component replies to the message from the environment and sends an reply message. This reply message is intercepted by the Monitor.
6. The Monitor forwards the intercepted message to the Validator.
7. The Validator checks that the message is correct with respect to the specification. If the message is not correct the Validator flags the message as an error to be handled by the system.
8. The reply message is sent to the environment.

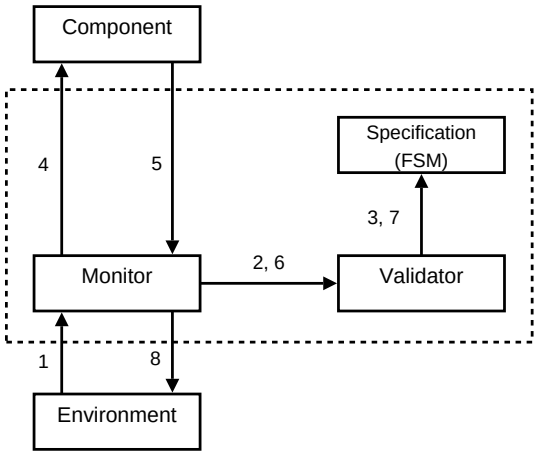

Figure 2. The validation engine.

3 Protocol Specifications

All our specifications describe protocols from the perspective of one component and we assume for simplicity that the component is the *server* in a client/server relationship, though the approach can be extended to more general architectures. In our setting a *client* is any component in the environment that communicates with the component. Although the environment may consist of more than one client, the specification only describes the correct communication between the component and one client at a time. (Multiple clients are considered below.) The specifications assume that the component is not active and only replies to requests from its clients. In the cases when the component makes active connections to other components in the environment, it takes the role of client with regard to the contacted component.

A protocol expresses what is considered to be a correct communication between a component and one of its clients. Protocols can specify different properties of the communication and they do not necessarily need to specify all aspects of the communication; underspecification is achieved by only considering certain events in a protocol specification. We want our protocols to express which messages the component accepts at a specific point in the run of a protocol and how it will reply to a message at that point, depending on the communication history of the component. These protocols can be intuitively expressed as *extended finite-state machines* (EFSMs). There are several different definitions of EFSMs. For our purposes we extend finite states machines with variables. An *EFSM* is defined as a tuple $M = (S, s_0, V, I, O, T)$, where S is a finite set of states; s_0 is a distinguished initial state, $s_0 \in S$; V is finite set of variables; I is a set of input symbols; O is a set of output symbols; and T is a set of transitions, $T \subseteq S \times I \times O \times S$. The state of an EFSM is an abstraction of the communication history, and the transitions from a state express which messages are accepted in that state; the input symbols are used to express what messages are accepted in a state, the output symbols will be used for error handling in Sec. 5.

A *variable* is a tuple on the form $(name, type, value)$ consisting of a unique *name* for the variable, the *type* of the variable, and a *value* from the domain of the variable type. We require that the domains of all types are finite to ensure that all EFSM can be translated into FSM. We will call a specific assignment of values to the variables V in an EFSM an *instance* of V. The notation V_i refers to instance i of V. An instance of V is always associated with a state in the state machine. An instance V_i of V and an associated state $s \in S$ is called a *configuration* of the state machine M.

Input symbols are tuples $(guard, message, pred)$, where *guard* is a predicate that must be $true$ in a state before a transition can be taken, and *pred* relates the states before and after the transition. A *guard* is a predicate over the current instance of V. Let *guard*(V_i) denote the evaluation of *guard* on the variable instance V_i. A *predicate transformer* is a relation between the variable instances before and after a transition. Let *pred*(V_i, V_j) denote the evaluation of the predicate transformer *pred* on the variable instances V_i and V_j. The predicate transformer *skip* evaluates to *true* if and only if all variables in the variables instances V_i and V_j have the same value.

If the guard of the input symbol of a transition evaluates to *true* for a given instance V_i (i.e., *guard*(V_i)), we call a transition *enabled*. A transition can not be taken unless it is enabled. When a transition is taken in a state s, the variable instance V_i associated with s is transformed into a variable instance V_j associated with the state after the transition. For all transitions that are taken the predicate transformer associated with the transition should always evaluate to true.

Protocol specifications based on regular expressions. Common ways of representing finite state machines, such as transition diagrams and state tables, are cumbersome for large specifications. Therefore, we have developed the specification language SSL, based on regular expressions, to make it easier to express protocol specifications. SSL supports variables, guards, predicate transformers, and alias definitions. An SSL specification can be automatically transformed into an EFSM [19]. Fig. 3 shows a simple SSL specification of a user manager component. The user manager allows users to login and, if a login is successful, returns a session identifier. The user manager also provides the possibility of checking if a session identifier is in use by the `isActive` operation, which returns `true` if the identifier is active and `false` otherwise.

SSL specifications are divided into three parts; the dec-

```
Set $ids = {}

alias success = return login(#val);
     check not(#val equals null);
     do #val add-to $ids
alias failure = return login(#val);
     check #val equals null

[[ login( #user, #pass ) : [ failure ] | [ success ] ]|
 [ isActive(#id); check #id in $ids :
   return isActive(#val); check #val equals true ]|
 [ isActive(#id); check not(#id in $ids) :
   return isActive(#val); check #val equals false ]]*
```

Figure 3. A specification of the user manager.

```
def performTransitions( message ):
  for state in currentStates:
    for transition in state.transitions:
      if message == transition.message
         and <transition is enabled>:
        <update variable instance>
        <add next state to newCurrentStates>
  if <newCurrentState is empty>:
    <found specification violation>
  <assign newCurrentState to current states for client>
```

Figure 4. Validation algorithm using an EFSM.

laration of variables, the definitions of aliases, and the description of valid message sequences. In Fig. 3 the variable `$ids` is the set of all active session identifiers. An alias defines a name for parts of the specification, allowing modularity. In the example specification we declare two aliases `success` and `failure`, respectively defining the criteria for successful and unsuccessful login. The last part of the specification is the description of valid message sequences. This part is based on regular expressions, with similar syntax and semantics. We use ':' to separate the messages, '*' as the Kleene closure operator, '|' as the union operator, and '[' and ']' as grouping symbols. In addition to common regular expression operators, guards and predicate transformers are supported with the `check` and `do` keywords respectively. The complete syntax and semantics of SSL can be found in [19].

Specification of multiple clients. The presented specification language does not explicitly distinguish different clients in protocols. In practise this is too restrictive as components often have many clients simultaneously in multithreaded programs. The specifications are now extended to support multiple clients.

Due to the potentially high number of possible communication interleavings, direct support for multiple clients in specifications is difficult and would tie the specification to a fixed number of clients, reducing flexibility. Instead, we support multiple clients through an on-the-fly transformation of the specification.

The interleaving semantics of parallel processes can be represented by using the *asynchronous product* of two EFSMs [12]. Using a standard definition of the asynchronous product, we obtain specifications that support the interleaving of communication from several clients. Whenever a new client starts communicating with the component, the product is performed on-the-fly. Notice that the communication with one client often affects the component communication with other clients, through some of the shared variables. For instance if a user manager component has given one client a session identifier, it should not give the same session identifier to another client. Shared variables can be automatically renamed to avoid name clashes if needed.

4 Validating Histories

The communications between a component and its clients are represented as sequences of messages, which must satisfy the protocol specification. We say that a message sequence *violates* the specification if no transitions are possible from the current state of the product EFSM for a given message. For simplicity, we limit our approach to *synchronous validation*, which means that when a message is intercepted and forwarded to the Validator for validation, the program blocks until the validation is completed.

Pseudo-code for an algorithm which does runtime validation for a given message is shown in Fig. 4. The algorithm tries all possible transitions from the current states of the specification machine when a message is received. The transitions are tried by iterating over all the current states of the specification machine (a non-deterministic machine can be in several states at once [13]). For each possible state, the algorithm iterates over all the transitions from the state and checks for enabled transitions. All enabled transitions are performed and the end state of every performed transition is put in a list of new states. A variable instance, updated to satisfy the predicate transformer, is associated to each state in the list. If no transition is enabled, the algorithm reports an error.

5 Runtime Error Handling

The runtime validation technique described in Sect. 4 is now extended to an approach for runtime error handling which works transparently for the monitored program. The approach works by replacing erroneous messages with corrected messages, based on the output from the specification machine. A conceptual view of the framework with this extension is shown in Fig. 5; the numbers in the following list correspond to the numbers in the figure:

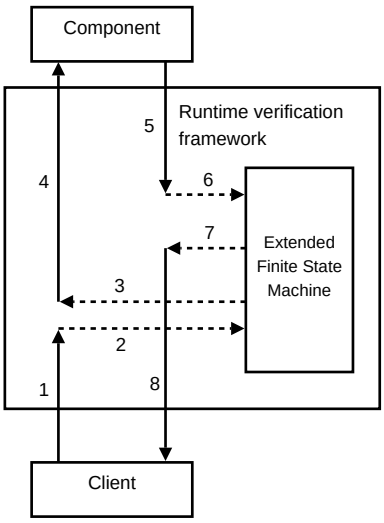

Figure 5. A conceptual view of the framework with the new specification model.

1. The client sends a message to the component.
2. The framework intercepts the message, which is forwarded to the EFSM.
3. The EFSM outputs a message, which need not be the message it received.
4. The framework receives the output message from the EFSM and sends it to the component.
5. The component emits a return message, addressed to the client.
6. The framework intercepts the return message and forwards it to the EFSM.
7. The EFSM outputs a message which need not be the message it received.
8. The framework receives the output message from the EFSM and sends it to the client.

Remark that performing error handling by transparently replacing messages con itself be a possible source of errors. Specific strategies are needed to handle errors correctly. One such strategy, which may be used to prevent a client from obtaining information to which it should not have access, is to replace sensitive message content by harmless information without breaking the logic of the protocol. An example illustrates how the strategy works. In a web-based banking application clients can access their account information and perform transactions, but only if they have been authorised first. A possible error in the banking application could enable an unauthorised client to access account information. By analysing the message sequences this may be discovered at runtime. In our example, if such an error is detected the specification machine replaces the sensitive

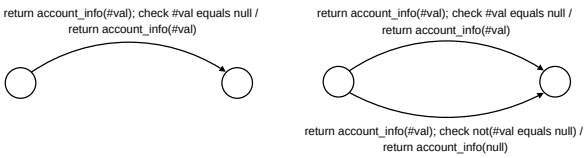

Figure 6. Error handling: A specification (left) is extended with error handling (right), where labels are on the form input/output.

account information with no information, thereby preventing leakage of the account information. More sophisticated error recovering strategies can be written with our approach.

To support this strategy, the specifications need to be slightly extended. For each error that we want to handle, a transition is added which takes the erroneous message as input and has the corrected message as output. In addition the transition should start and end in the same states, as the transition having the correct message as input and output. For instance in the web banking example, one transition would have a message with no information as input and the same message as output, the error handling transition would have a message with information as input and a message with no information as output. Fig. 6 shows a part of a specification with and without error handling. Not all errors can be handled by replacing message content, so other strategies are also needed. Some other strategies are discussed in [19].

6 A Prototype Implementation

A prototype of the approach has been implemented, which monitors and validates Java programs [19]. This section briefly describes some implementation decisions. The prototype represents components by single objects, method calls and method returns by messages, and clients by threads. The last choice is because Java does not provided access to unique object identifiers, while thread identifiers are unique in practise. The architecture of the prototype tool is shown in Fig. 7 and consists of the following components:

- The *specification parser* reads an SSL specification and generates an EFSM, stored in XML format [20].
- The *monitor generator* generates a specific monitor for the given objects and SSL specification.
- The *monitor*, generated by the monitor generator, monitors messages to be sent or received and forwards intercepted messages to the Validator.
- The *validator* validates the sequence of messages received from the monitor with respect to the EFSM specification. To avoid race conditions, only one message can be validated at the time.

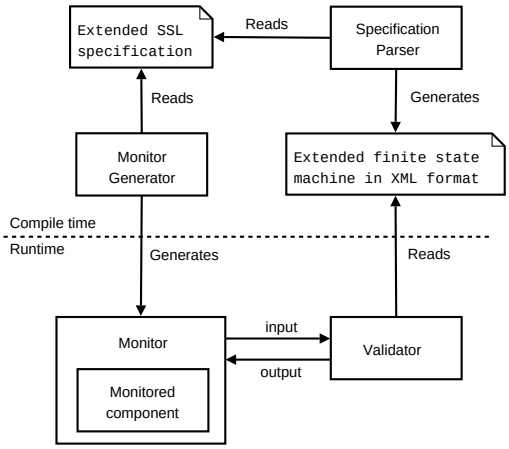

Figure 7. Architecture of the tool.

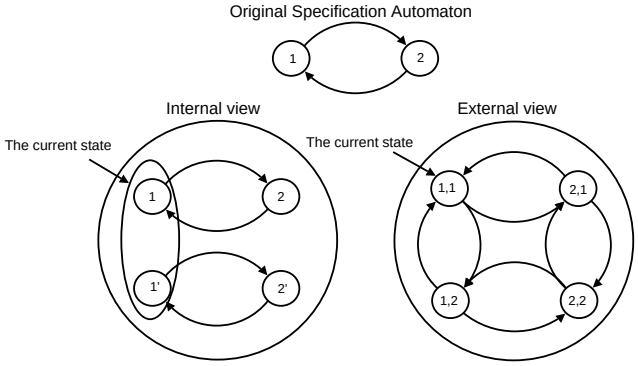

Figure 8. Asynchronous product.

Only the monitor and the validator are active at runtime. The specification parser and the monitor generator are only used before program execution.

The monitoring should be *transparent* to both the component and its clients. This means that no code in the component nor its clients should need to take the monitoring into account. Consequently, we can retrofit the program either at the source code level, the byte code level, or the interpreter level. We have chosen to do so at the byte code level as there are several tools that support changing the Java byte code; e.g., AspectJ, BCEL, and ASM[1]. For the prototype, we follow an aspect oriented approach, using AspectJ [8, 15].

Implementation of the asynchronous product. The asynchronous product operation results in specifications that support the interleaving of multiple client threads, but the number of states in the specification machine can grow significantly as the number of clients increases. The validation of the communication protocols of a component with multiple clients therefore requires some optimisation to reduce the number of states. For this purpose the asynchronous product of specification EFSMs is not done explicitly. Instead a copy is kept of each EFSM specification, and the global state is thus defined by the set of variables, together with a tuple that contains the current state of the EFSM for each client. When a message is received from a client, a transition is performed in the EFSM associated with that client. The approach is illustrated in Fig. 8, which shows the original specification, the internal view of the implementation, and how the implementation looks from the outside. Internally, there are two copies of the EFSM specification and the current state of the product state machine is a tuple with one state from each copy of the EFSM.

Monitoring method calls. In Java, the message can be recorded at two different moments, which for the purpose of monitoring are not equivalent: either when the method is called or when the method starts executing. This difference arises because of Java's synchronisation mechanism[2]. The two choices for monitoring correspond to recording the call before or after the synchronisation. The first choice can lead to validation of the wrong message sequence, as shown in the following example. Let A, B, and C be three clients communicating with a synchronised component D. First A calls a method in D. Since the synchronisation lock is free, A is given access. Then B calls a method in D, but B must wait because the synchronisation lock is taken. Then C calls a method in D and C must also wait for the lock. When A finishes and frees the lock, Java does not guarantee that B will be given access first. If C is given access first and the call message was recorded at call time, there will be a mismatch between the message sequence seen from the validation point of view and the message sequence seen from the component's point of view. To avoid this mismatch the monitoring is performed just before a method starts executing and not when the method is called.

7 Examples and Benchmarks

We have run some benchmarks for the prototype implementation on two Java examples. The first example is an implementation of a user manager component (see Fig. 1). The user manager is a component which performs authentication in web applications. The second example is an implementation of a query manager component for use in desktop applications. The query manager component has access to a database and all queries to the database pass via the query

[1] See `www.eclipse.org/aspectj`, `jakarta.apache.org/bcel`, and `asm.objectweb.org`, respectively.

[2] A call to a synchronised method in Java can only be executed by a thread if no other thread concurrently executes a synchronised method on the same object.

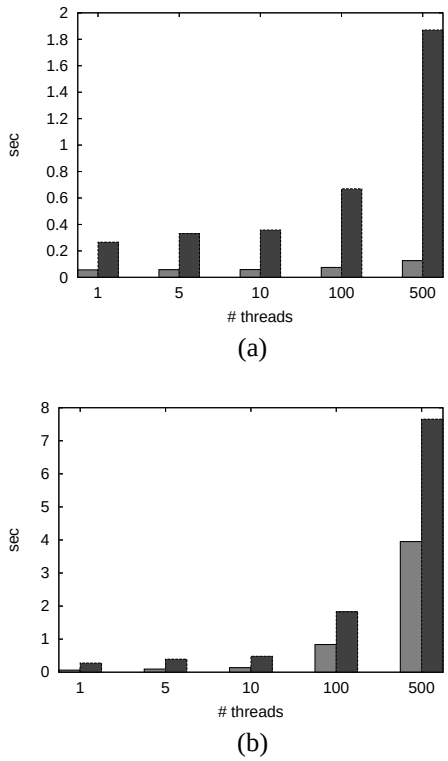

Figure 9. Benchmark results for (a) the user manager and (b) the query manager. The dark columns include monitoring.

manager. To speed up certain queries, the query component stores the result of the last query for each of its clients. This stored result can then be manipulated locally to get the desired view of the result. Since the query manager can avoid making some of the queries to the database, the response time is reduced and so is the load on the database.

The *benchmark* is based on the Unix *time* command, which executes other programs and records different statistics about the executions. We use the *user* statistic to record the time it would take a program to execute if no other programs where using the CPU. This benchmark is simple and gives consistent results when run on the same program several times.

The benchmark has been applied to both examples with and without monitoring. Each example was run with 1, 5, 10, 100, and 500 threads. To ensure that one divergent result should not have a large impact on the final result, the benchmark was run 50 times for each case. The benchmark results for the user manager component are shown in Fig. 9 (a) and the benchmark results for the query manager in Fig. 9 (b).

For the user manager there is a significant increase in execution time when the component is monitored, which becomes more prominent when the number of threads increases. This is probably because the validator only validates one message at a time. The validator uses a lock to ensure mutual exclusion; since all the threads need to access the lock, it is likely that threads must wait before acquiring the lock. A more sophisticated locking scheme may therefore increase the performance significantly. For the query manager, the execution time only increases by a factor of 2 or 3 with monitoring. The main difference between the query and the user manager is that the query manager methods are significantly slower than the user manager methods; the time used to acquire the lock compared to the method execution time will therefore be less significant than for the user manager component.

These benchmarks suggest that an approach using EFSM-based specifications can be implemented efficiently for runtime validation even for specifications that must handle a large number of clients.

8 Related and Future Work

The Design by Contract technique was introduced by Meyer for the Eiffel programming language [17] in order to allow runtime checks of specification violations and their treatment. This approach has been developed for Java in, e.g., JML [16] and Bandera [5]. JML can be used to model the behaviour of components, based on the guards and predicate transformers of methods. These specifications are compiled into runtime checks. Jass [2] extends guards and predicate transformers with trace assertions, expressed in terms of CSP processes [11]. Trace assertions are compiled into the Java program such that violations result in Java exceptions. Jass does not support trace assertions for multithreaded programs. A JML extension to support the specification of the order of communication has recently been proposed [4], but does not generalise to multithreaded programs. Multithreaded programs are difficult to handle in the JML setting because the guard may be broken before the method executes [18]. While these approaches require access to the component source code for compilation, our approach deals with components for which the source code is not available. Furthermore, the above-mentioned difficulties do not apply to our approach as guard and predicate transformer are associated with the sending of messages, which are atomic actions.

In contrast to Java-MOP, which focuses on providing a platform for supporting various kinds of program monitoring and analysis at runtime, our approach is more focused on specification. Java-MOP supports runtime error handling through user specified violation handlers, while we support error handling at the specification level, allowing the error handling to be completely transparent for the monitored program. As stated in the introduction we do not require knowledge of the internal logic of the components and

we only concentrate on a component external behaviour, in contrast to JPAX, which is based on the monitoring of state variables [10].

The use of automata-based specifications for component behaviour is not new, e.g., [6] and [1]. The latter work introduces the notion of interface automata, which could be used instead of the I/O automata we use in our approach.

This paper proposes runtime verification based on communication histories. Several interesting research topics remain to be addressed, including:

- *More efficient monitoring.* To perform runtime verification on deployed software, the monitoring and verification should not interfere unreasonably with normal system activity. More efficient monitoring and validation may contribute to the success of runtime verification.
- *More sophisticated monitoring.* Our approach might be extended to monitor security properties such as, e.g., *admissibility* [9]. It would also be interesting to monitor more advanced protocol specifications such as, e.g., assume-guarantee history specifications [14].
- *JML integration.* It seems that our approach for specifying communication protocols could, at least partially, be integrated into JML. This would for instance give the approach access to a more refined type system and a richer language for specifying guards and predicate transformers, thereby increasing the expressibility of the approach.

9 Conclusion

In this paper we have presented an approach to runtime verification based on the specification and validation of history-sensitive communication protocols for object-oriented components. The specification of communication protocols was written in a simple specification language based on regular expressions and represented as extended finite state machines, letting the state of the machine be an abstraction of the history. By representing the protocols as extended finite state machines the protocols could be expanded on-the-fly to support interleaving communication to the multi-threaded case with any number of clients, thereby transferring the difficult task of specifying all possible interleavings to the framework implementation. We have also presented a prototype implementation of the approach and performed two benchmark tests to argue for its feasibility in practise.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[2] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with assertions. *ENTCS*, 55(2), 2001.

[3] F. Chen and G. Rosu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005.

[4] Y. Cheon and A. Perumandla. Specifying and Checking Method Call Sequences of Java Programs. Technical Report 05–36, Dept. of Computer Science, Univ. of Texas at El Paso, Nov. 2005.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the Bandera specification language. *International Journal on Software Tools for Technology Transfer*, 4(1):34–56, 2002.

[6] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[7] B. Eckel. *Thinking in Java*. Prentice Hall, 3rd edition, 2003.

[8] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing Aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001.

[9] P. Giambiagi and M. Dam. On the secure implementation of security protocols. *Sci. Comput. Program.*, 50(1-3):73–99, 2004.

[10] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Form. Methods Syst. Des.*, 24(2):189–215, 2004.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Int. Series in Computer Science. Prentice Hall, 1985.

[12] G. J. Holzmann. Software model checking with Spin. *Advances in Computers*, 65:77–108, 2005.

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, Addison-Wesley, 2nd edition, 2001.

[14] E. B. Johnsen, O. Owe, and A. B. Torjusen. Validating behavioral component interfaces in rewriting logic. In *Proc. FSEN 2005*, volume 159 of *ENTCS*, pages 187–204. Elsevier, May 2006.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with ASPECTJ. *Communications of the ACM*, 44(10):59–65, 2001.

[16] G. Leavens, A. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[17] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[18] E. Rodríguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP'05*, volume 3586 of *LNCS*, pages 551–576. Springer, 2005.

[19] Ø. Torget. Runtime Validation of Communication Histories: An Automata-based Approach. Master's thesis, Dept. of Informatics, University of Oslo, May 2006.

[20] XML Extensible Markup Language 1.0, W3C Recommendation. Available online: `http://www.w3.org`, Feb. 2004.