From Object-Oriented Code with Assertions to Behavioural Types

Cláudio Vasconcelos
NOVA LINCS and DI-FCT, Univ. NOVA de Lisboa

António Ravara
NOVA LINCS and DI-FCT, Univ. NOVA de Lisboa

ABSTRACT

The widespread use of service-oriented and cloud computing is creating a need for a communication-based programming approach to distributed concurrent software systems. Protocols play a central role in the design and development of such systems but mainstream programming languages still give poor support to ensure protocol compatibility. Testing alone is insufficient to ensure it, so there is a pressing need for tools to assist the development of these kind of systems. While there are tools to verify statically object-oriented code equipped with assertions, these mainly help to prevent runtime errors. However, a program can be ill-behaved and still execute without terminating abruptly. It is important to guarantee that the code implements correctly its communication protocol. Our contribution is a tool to analyse source code written in a subset of Java, equipped with assertions, and return it annotated with its respective behavioural types that can be used to verify statically that the code implements the intended protocol of the application. A running example illustrates each step of the tool.

Keywords

Assertions; behavioural types; object-oriented programming

1. INTRODUCTION

Communication protocols play a central role in the design and development of distributed concurrent software systems, which are these days communication-based, not only due to running in multi-core architectures, but also due to the widespread use of service-oriented and cloud computing.

Concurrent programming is very challenging; developing distributed protocol-based applications is not only quite elaborat, but also error-prone due to how hard it is to reason about the behaviour of a multi-threaded program [14, 9].

SAC'17, April 3-7, 2017, Marrakesh, Morocco
Copyright 2017 ACM 978-1-4503-4486-9/17/04...$15.00
http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

As Dijkstra stated, "Program testing can at best show the presence of errors but never their absence" [6, 7]. It is necessary to develop techniques that help to create safe and well-behaved systems, such as tools to assist us while provably providing correctness guarantees, or extensions of programming languages with constructs to specify the intended behaviour and to ensure that it is followed. It is important that these techniques allow programmers to test their systems while developing them instead of testing only at the end, which can be complicated.

There are many approaches to guarantee code correctness but we are interested in analysing the source code and assisting the developer by identifying errors in the development phase. We are also interested in more than safety properties, as the absence of run-time errors does not imply that an application is well-behaved. Consider, for instance, an API to manipulate files, providing functionalities to open, close, read, write, and test emptiness. Obviously, there is a usage protocol underlying such API: on first opens the file; before reading one should test for (non-)emptiness, and once the work is done, one should close the file. Guaranteeing (statically) that the code never goes wrong does not ensure its "usefulness". An important (liveness) property is that client code using the API should always follow its protocol.

The aim of our work is to bridge the world of programming in Java with assertions[1] with the world of behavioural typed programming [13, 1], in particular in Java. While the former is becoming increasingly popular and well supported (being part of the Java language for more than a decade now), the latter has a growing impact and will probably soon be incorporated into mainstream languages. We argue that programming using behavioural types can be, in most cases, easier than programming with assertions, because they can be more intuitive to write, informative and easier to check, but right now the properties guaranteed by type systems and assertions complement each other. For example, current type systems do not guarantee the absence of NullPointerExceptions.

We developed an algorithm to convert Java with assertions in a form of behavioural types (henceforth called *usage*, a textual representation of a finite automata). Usages represent all the safe sequences of method calls and are (enhanced forms of) class types, checkable at compile-time.

[1]http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html

In a nutshell, our approach is the following: given a program written in a subset of Java, fully annotated with assertions that we consider correct, returns its usage. The goal is to provide developers with abstractions extracted from the code to represent its behaviour. Furthermore, these abstractions can be attached to the code and statically verified, ensuring also its correctness.

2. RELATED WORK

We build on a consolidated body of work on behavioural static analysis of concurrent systems.

2.1 Behavioural types and type systems

Behavioural types allow the specification of interaction patterns of processes through expressive type languages. In behavioural type systems, the type-checker can statically verify these behavioural types and prove that these interaction patterns they specify are safe [13].

2.1.1 Typestates

Typestates are an extension of the concept of type that makes it possible to define which operations are allowed in a particular context, helping to detect at compilation time, particularly during type-checking, unexpected sequences of operations (like reading a file after closing it) [17].

Damiani *et al.* propose to include in Java-like languages a feature to declare state dependent object behaviour [5]. Using a type and effect system, the authors ensure that, during the execution of a method call there are no re-entrant calls nor access to un-initialised fields.

Garcia *et al.* presented the concept of typestate-oriented programming, which consists on the extension of object-oriented programming with the notion of typestate [10]. In typestate-oriented programming each state is represented by a class, with each of these classes having their own representation and methods. In this context, the class of the object represents its typestate, which can dynamically change during runtime. Aldrich also developed the Plaid language, a typestate-programming language that uses these foundations as its core [16].

2.1.2 Session types

Session types are an instance of behavioural types developed by Honda *et al.* [12], aiming at guaranteeing consistent communication patterns never leading to runtime errors.

Session types were neatly integrated into object-oriented programming languages (as shown in a recent report [1]).

2.2 Behavioural type inference

There are several proposals that aim for session-type inference of programs written without explicit session types. Hüttel *et al.* present an approach to session type inference for the π-calculus based on constraint generation and solving [11], but they argue that it should be possible to adapt the work for other programming languages since these constraints are present in other languages with binary session types. Previously, Padovani showed how to encode session types into linear types and defined a type reconstruction algorithm [15].

Caso, Braberman, Garbervetsky and Uchitel present the idea of generating abstract behavioural models, which are similar to typestates [3]. We use the latter approach as one of the basis of the work presented herein.

3. THE MOOL LANGUAGE

The Mool programming language is a Java-like language that integrates a form of behavioural types based on session types into an object-oriented language [2]. In Mool each class has a behavioural type (called a usage) associated that specifies safe orderings of method calls. A static type-checking system verifies if the code adheres to the usages of the classes.

We want to infer the usage from the source code, instead of making the programmer write it for each class. Although we leave formal proofs for future work, the tool developed produce usages that, if associated to the classes of the analysed source code, should type-check in the Mool compiler.

The target language for our behavioural type inference tool is a variation of Mool which we call Mool$^-$.[2] It differs from the original version in two aspects: (1) it is based on a revised version of the language [18] that sol some bugs and extends expressiveness, and (2) differs from that version by not having usages and instead allowing to annotate classes with assertions. With these assertions, programmers can specify the expected state of an object during its existence through invariants, and also specify the state of the object before and after a method execution.

4. BLOG EXAMPLE

The running example we use herein is based on a blog scenario. Due to the lack of collections in Mool (it is a proof-of-concept language), we assume that this blog can contain only one post. There are two types of users:

Admin Can create, remove, and publish a post. The admin can remove a post only if it has not been published before.

Viewer Can view a published post.

Listing 1 shows the `Post` class. Since the access control to a `Post` object is done by the `Blog` object, it can have its only method available at any time, which explains why every assertion is `true`.

Listing 2 shows the `Viewer` class. When initialised, a session is created and it can be closed at any time. The viewer must request the post, which will save the post locally. The viewer must be able to request a post right after initialisation.

Listing 3 shows the `Blog` class. When a blog is initialised it does not have a post so a new one must be created. After creating a new post it can be removed or it can be published, making it public to the viewers and blocking any write operation on it.

[2]The syntax of Mool$^-$ is available at http://usinfer.sourceforge.net/mool_minus_syntax.pdf

```
class Post {

  string title; string body;

  //@ invariant true;
  //@ initial true;
  void Post(string t, string c) {...}

  //@ requires true;
  //@ ensures true;
  string readPost() {...}
}
```

Listing 1: Code for the *Post* class

```
class Viewer {

  Blog blog; Post post;
  boolean has_post; boolean session;

  //@ invariant true;
  //@ requires b.postIsPublic();
  //@ initial post == null && blog != null && session
      ;
  void Viewer(Blog b) {...}

  //@ requires !has_post && session;
  //@ ensures (blog != null && session) && (|result|
      -> post != null);
  boolean requestPost() {
    if(blog.postIsPublic()) {
      post = blog.viewPost(); true;
    } else {...}
  }

  //@ requires post != null && blog != null &&
      session;
  //@ ensures post != null && blog != null && session
      ;
  void readPost() {...}

  //@ requires blog != null && session;
  //@ ensures blog != null && !session;
  void endSession() { session = false; }
}
```

Listing 2: Code for the *Viewer* class

```
class Blog {

  Post post; boolean is_public;

  //@ invariant true;
  //@ initial !is_public && post == null;
  void Blog() { ... }

  //@ requires !is_public && post == null
        && p != null;
  //@ ensures !is_public && post != null;
  void newPost(string title, string body) { post =
      new Post(title, body); }

  //@ requires !is_public && post != null;
  //@ ensures !is_public && post == null;
  void deletePost() { post = null; }

  //@ requires true;
  //@ ensures |result| -> post != null;
  boolean hasPost() { post != null; }

  //@ requires true;
  //@ ensures |result| -> is_public;
  boolean postIsPublic() { post != null; }

  //@ requires is_public && post != null;
  //@ ensures is_public && post != null;
  sync Post viewPost() { ... post; }

  //@ requires !is_public && post != null;
  //@ ensures is_public && post != null;
  void publishPost() { is_public = false; }
}
```

Listing 3: Code for the *Blog* class

5. USAGE INFERENCE TOOL

In this section we describe the usage inference tool, adapted to our target language. This tool receives a program composed by classes written in Mool$^-$ and returns those same classes as code written in Mool, *i.e.*, without assertions and annotated with their respective usages. The tool works through three stages, with the first two based on existing work, which we adapted to fulfil the requirements of our target language. The first stage extracts the typestates of each Mool$^-$ class, which are then translated into usages in the second stage, and finally the third stage defines the usage state each object starts in. For each stage we provide the output for the example presented in section 4 to exemplify what to expect from each. We make available the full ML code of each stage of our algorithm.[3]

5.1 Stage 1: Typestate generation

In this first stage of the usage inference tool we use the behavioural model approach of Caso *et al.* [3]. This approach consists on an algorithm that receives the source code of a program equipped with assertions that represent invariants and requires clauses and constructs automatically a enabledness-preserving behaviour model, which is similar to a typestate. These behaviour models are permissive, meaning that they include every possible operation sequence of the program. The authors of the original algorithm we adapted argue that their technique ensures that the constructed behaviour models are always permissive independently of the library's internal state being finite or not.

We made a few changes to the algorithm to fit it to our target language. The first concerns checking the validity of the assertions. In their presentation of the algorithm [3], the authors suggest using code reachability to do the required validity checks, arguing that their implementation does not use a theorem prover due to the lack of postconditions. We, however, decided to include postconditions in our language and the init clause that represents the state of the object after its initialization. Therefore, we are able to use a theorem prover (Z3[4]) to check the validity of the assertions in first-order logic.

The second important change is related to non-determinism. The original algorithm allows non-deterministic transitions, depending on the internal state of the program. Since, as usual in a programming language, Mool does not support non-determinism, we also adapted the original algorithm in this respect. Mool allows transitions to have, at most, two target states, with these being based on choice, which are represented in usages through variant types. These transitions can only be triggered by a boolean method and they rely on its result to choose the next state, hence the two target state limit for each transition.

For the algorithm to allow transitions based on choice (according to the result of a boolean method), it needs to know what state to choose if the result is true and what state to choose if the result is false. For example, consider a File class that follows the protocol defined in Figure 5.1. The transi-

[3]https://sourceforge.net/p/usinfer/code/ci/master/tree/ml/.

[4]https://github.com/Z3Prover/z3/wiki

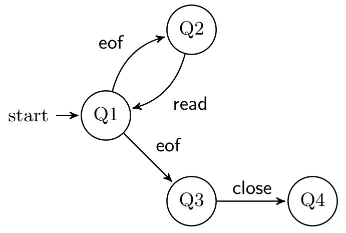

Figure 5.1: Non-deterministic behaviour a *File* class

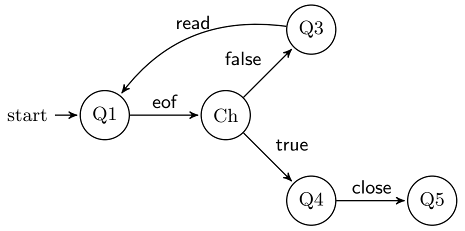

Figure 5.2: Behaviour a *File* class

tion relation δ is expected to include the triples (Q1,eof,Q2) and (Q1,eof,Q3), since δ needs to include a transition corresponding to the case when the method eof returns true and another transition corresponding to the case when the method eof returns false. We explicitly need to know which target state corresponds to what result since we cannot rely on the order of how they are presented (in the previous example, the choice that corresponds to the false result is presented first). In our algorithm the transition relation is as function defined as in Figure 5.2.

Such transition function is defined by the object state that causes the method to return true and the object state that causes the method to return false. So, in these situations, the post-condition of the method that triggers the transition must specify both states.

Therefore, our algorithm returns, for each class, a state machine representing its typestate. Figures 5.3, 5.4 and 5.5 show a graphical representation of the generated typestates for the classes of the example presented in Section 4.

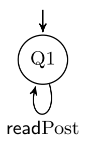

Figure 5.3: Typestate for the *Post* class

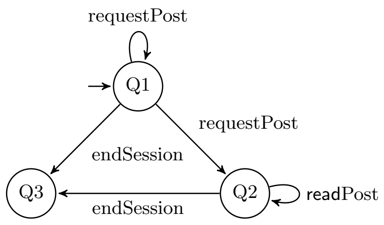

Figure 5.4: Typestate for the *Viewer* class

The state machines returned by the first phase of the algorithm represent all the possible sequences of method calls of a given class. We still need to convert them in *usages*, the syntactic terms used as class types in Mool, which are like regular expressions. We describe this step in the following section.

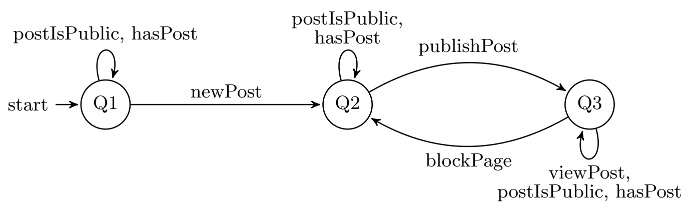

Figure 5.5: Typestate for the *Blog* class

```
usage lin{Post; Q1} where
  Q1 = un{readPost; Q1};
```

Listing 4: Code for the *Post* class usage

5.2 Stage 2: Usage generation

The second stage of our algorithm consists thus on obtaining usages from the typestates obtained from the first phase. To do this we will use the idea presented by Collingbourne and Kelly [4], which consists on a three-stage algorithm that receives code from a very simple language similar to C and produces a session type from it.

Although the technique itself infers session types directly from code, it is defined to work on a limited language that lacks object and concurrency support, two important aspects of our target language. If we were to adapt it to our target language we would need to extend it with the missing features. Instead of doing such extension, we adapted the algorithm of Caso *et al.* [3].

Therefore, we are only going to use the third stage of Collingbourne and Kelly's algorithm, where it receives a state machine, converts it into a deterministic state machine, and applies a function that translates it into a session type. Since Mool has usages and not session types we need to adapt the function so that it translates state machines to usages instead. However, the conversion of non-deterministic state machines to deterministic ones, in our case that, instead of non-determinism in supports choice, would allow unwanted behaviour. Instead of following Collingbourne and Kelly's approach in this respect, we need to introduce variant types. Determining which target transition corresponds to which boolean value is done using the information given by the new transition function presented in the previous stage.

Finally, since Mool supports shared and linear objects, the latter with behaviour captured in usage types and the former with behaviour captured in "standard" class types, we added a mechanism that ensures the generated usage does not have transitions that go from a shared state to a non-shared state or a non-equivalent non-shared state, *i.e.*, a non-shared state that offers a different set of methods.

In short, this second phase of our algorithm is loosely inspired by that of Collingbourne and Kelly, but has important modifications. To illustrate this phase of our algorithm, consider listings 4, 5 and 6. Each presents the usage type corresponding, respectively, to the typestates in Figures 5.3, 5.4, and 5.5. These usages were obtained by the algorithm from the typestates generated in the previous phase of the algorithm.

```
usage lin{Viewer; Q1} where
  Q1 = lin{endSession; end +
    requestPost; <Q2 + Q1>}
  Q2 = lin{endSession; end +
          readPost; Q2};
```

Listing 5: Code for the *Viewer* class usage

```
usage lin{Blog; Q1} where
  Q1 = lin{postIsPublic; Q1 + hasPost; Q1 + newPost;
      Q2}
  Q2 = lin{publishPost; Q3 + postIsPublic; Q2 +
      hasPost; Q2 + deletePost; Q1}
  Q3 = un{viewPost; Q3 + postIsPublic; Q3 + hasPost;
      Q3};
```

Listing 6: Code for the *Blog* class usage

5.3 Stage 3: Object usage state inference

The previous stage generated usage types for classes. We still need to generated usage types for the declarations of fields, parameters and method return types, when these manipulate objects.

Mool offers the possibility of indicating the usage state an object starts in its declaration. Consider the following excerpt of the Viewer class:

```
class Viewer {
  ...
  Blog[Q3] blog;
  ...
}
```

We want the blog field to be initialised in a state where the viewer can request a post right away which, according to the generated usage for the Blog class in listing 3, corresponds to the usage state Q3, as indicated in the declaration of the blog field.

In the context of our work, we do not expect the programmer to know beforehand the states that will compose the generated usage, so the programmer does not have a way to indicate the usage state of an object when initialised. Although, we can expect the programmer to know the overall state of an object when initialised and be able to express it through assertions.

It is possible to express the expected state of the instance received as a parameter in the precondition of the method. In this case, since the field blog is initialised in the constructor with an object passed as an argument, we can specify the usage state of blog by defining the constructor as follows:

In the requires clause we call the method postIsPublic on the parameter b, stating that b must have the post available for the viewer.

In Mool one can also define the usage state of an object returned by a method. The method viewPost of the Blog class returns the field post, an instance of the Post. Since we want the object post to be initialised, we should specify the return type of the method as follow:

```
sync Post[Q1] viewPost() {
  ...
  post;
}
```

This usage state can be inferred using the method postcondition, where it is possible to express the expected state of the returned object. For this example, since the usage state Q1 corresponds to the state where the object of type Post is initialised, we can just specify that the returned object is not null.

```
//@ requires is_public && post != null;
//@ ensures is_public && post != null;
sync Post viewPost() {
  ...
  post;
}
```

All these tasks are done by an algorithm that goes through all the methods of each class and does the following:

Step 1 Checks for parameters containing objects. The algorithm uses the precondition of the method to determine their state. In this context only the premises related to the parameters being verified are considered.

Step 2 Checks the return type of the method. If it is a class, the algorithm uses the postcondition of the method to determine the usage state of the return type. Again, only the premises related to the class field or local variable being returned by the method are considered.

Step 3 Analyses the code of the method and checks where are the fields initialised. For every initialisation value, the algorithm sets the usage state of the field with the same usage state of the value. Moreover, since the object used as the initialising value can be manipulated before being assign to the field, the algorithm also checks the calls on that object and keeps track of the its current usage state.

The final result of the algorithm is thus usage types for each class and each field, parameter, or result class type, in the form used by Mool[5].

One can thus then test the correctness of the resulting code using Mool's type-checking system, looking for, apart from data-errors, flow errors like protocol compatibility, completion, and null de-referencing. Notice that if the program type-checks, the program is provably free of such errors (Mool's type system is safe).

6. CONCLUSIONS AND FURTHER WORK

In this paper we present a behaviour type inference approach for a Java-like language called Mool. The reason we we choose to work with a small language instead of standard Java because right now behavioural type do not cope with features such as generics and collections. The algorithm takes a program fully annotated with assertions[6] and either fails: the code is *not well-typed* (in the standard sense) or it may produce a run-time error due to calling methods in an *incorrect order*; or returns a new version of the code with the classes annotated with behavioural types (called usages).

[5]A more detailed execution of the algorithm is available at http://usinfer.sourceforge.net/algorithm_details.pdf.

[6]We check with Z3 that the assertions are logically valid.

Usage types can then be statically checked to verify if the code is data-safe and flow-safe: there will be no (null pointer) exceptions, no methods called when they are not supposed to, and moreover, that the protocols of critical resources are fully executed. In a nutshell, usage types ensure *safe interoperability*, which in this case means object compatibility: all inter-object method calls are valid and happen at a time where the state of the object allows those calls.

Notice that statical behavioural type-checking is a way of automatically ensure the correctness of the assertions (thus obviating the burden of using Hoare logic to manually – even if machine-assisted – do it). Nonetheless, as usual in static analyses methods, our approach is incomplete. In some cases, the assertions may be correct but type-checking fails. the "problem" is that assertions talk explicitly about state but usages only refer it implicitly. We need the variant types to link state-based decisions with typestates. So, if the assertions are not informative enough, we infer a wrong or incomplete usage and type-checking fails.

We implemented the algorithms described herein, making them available in Sourceforge[7], along with a set of four examples (FileReader, Auction, Petition and Blog) we used to test it.The tool starts by generating a state machine representing a typestate, based on the assertions on the code. The tool then translates the generated typestates into usages. In the end, it defines the usage state that each object of the class starts with by using the assertions and the usages obtained in the previous stage. This tool is composed by three algorithms, with the first two adapted from algorithms presented in other works [3, 4], and the third one being original.

Future work will include developing correctness proofs for the algorithms (the ML version, being purely functional, offers a good basis for machine-assisted proofs using Why3 [8]) and automatically inferring assertions to the code (we are interested in postcondition inference using Hoare logic).

7. ACKNOWLEDGEMENTS

This research was partially supported by UK EPSRC grant EP/K034413/1 From Data Types to Session Types: A Basis for Concurrency and Distribution, and by FCT/MCTES under NOVA LINCS strategic project UID/CEC/04516/2013.

8. REFERENCES

[1] D. Ancona, V. Bono, M. Bravetti, J. Campos, P.-M. Deniélou, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, F. Montesi, R. Neykova, V. T. Vasconcelos, and N. Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2–3):95–230, 2016.

[2] J. Campos. Linear and shared objects in concurrent programming. Master's thesis, Univ. of Lisbon, 2010.

[3] G. D. Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Enabledness-based program abstractions for behavior validation. *ACM Transactions on Software Engineering and Methodology*, 22(3):1–46, 2013.

[4] P. Collingbourne and P. H. J. Kelly. Inference of session types from control flow. *Electronic Notes in Theorectical Computer Science*, 238(6):15–40, 2010.

[5] F. Damiani, E. Giachino, P. Giannini, and S. Drossopoulou. A type safe state abstraction for coordination in Java-like languages. *Acta Informaticæ*, 45(7-8):479–536, 2008.

[6] E. W. Dijkstra. A.M. Turing Award Winner. http://amturing.acm.org/award_winners/dijkstra_1053701.cfm.

[7] E. W. Dijkstra. Archive: The Humble Programmer (EWD 340). https://www.cs.utexas.edu/ EWD/transcriptions/EWD03xx/EWD340.html.

[8] J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming Languages and Systems (ESOP'13)*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[9] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'10)*, pages 221–230. IEEE Press, 2010.

[10] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *Transactions on Programming Languages and Systems*, 36(4):1–44, 2014.

[11] E. F. Graversen, J. B. Harbo, H. Hüttel, M. O. Bjerregaard, N. S. Poulsen, and S. A. Wahl. Type inference for session types in the π-calculus. In *Web Services, Formal Methods, and Behavioral Types - Revised selected papers of WS-FM'14 and WS-FM/BEAT'15*, volume 9421 of *Lecture Notes in Computer Science*, pages 103–121. Springer, 2016.

[12] K. Honda. Types for dyadic interaction. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[13] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Computing Surveys*, 49(1):1–36, 2016.

[14] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, pages 329–339. ACM, 2008.

[15] L. Padovani. Type Reconstruction for the Linear π-Calculus with Composite Regular Types. *Logical Methods in Computer Science*, 11:1–45, 2015.

[16] The Plaid Programming Language. http://www.cs.cmu.edu/˜aldrich/plaid/.

[17] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions in Software Engineering*, 12(1):157–171, 1986.

[18] C. Vasconcelos and A. Ravara. A revision of the Mool language. 2016.

[7]http://usinfer.sourceforge.net.