Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Design and Implementation of a Behaviorally Typed Programming System for Web Services

Filipe David Oliveira Militão (26948)

Lisboa
(2008)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Design and Implementation of a Behaviorally Typed Programming System for Web Services

Filipe David Oliveira Militão (26948)

Orientador: Prof. Doutor Luís Caires

Júri

Presidente:

- **Doutor José Alberto Cardoso e Cunha**, Professor Catedrático, Departamento de Informática da Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

Vogais:

- **Doutor Francisco Martins**, Professor Auxiliar, Departamento de Informática da Faculdade de Ciências, Universidade de Lisboa.
- **Doutor Luís Manuel Marques da Costa Caires**, Professor Associado, Departamento de Informática da Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do Grau de Mestre em Engenharia Informática.

Lisboa
(2008)

Acknowledgements

This work was partially supported by a CITI/PLM/1001/2007 research grant.

Summary

The growing use of the Internet as a global infrastructure for communication between distributed applications is leading to the development of a considerable amount of technologies to ease the deployment, description and data exchange among services and thus improve their interoperability. There is also a growing interest in the use of the "software as a service" business model where a software vendor develops and hosts applications to be used by its clients over the Internet. The use of these Web Services is provided through an API describing the interface of the service that can hide how the service provider hosts the application. This approach allows for the creation of an abstraction layer that offers additional capabilities without increasing the maintenance cost usually linked to the management of those machines (like software and hardware updates or just application/system configuration).

However, the main tools provided by the standards and existing technology to combine these services usually only account for limited automatic verification techniques (based on standard signature checking of methods in interface descriptions) and thus relying the behavioral compatibility among services to the programmer. The programmer then becomes dependent on the quality of the documentation and the development time available to manually (and without formal guarantees) assure the correctness of the code.

In this thesis, we propose a behavioral type system, in the context of **yak**, a prototype scripting language for web services, that enhances traditional typecheckers by allowing to statically check the correct usage of services (as remote or local objects). Our language uses behavioral annotations in the protocol descriptions, similar to regular expressions, that are translated to deterministic finite automatons during the typechecking phase. The intent of this work is to ease the creation and deployment of Web Services by providing a friendly integration of behavioral type concepts within a practical programming language, so to make the use of these services (with behavioral descriptions) transparent and effortless to the programmer. We also provide a full implementation of the interpreter, behavioral typechecker and run-time support system for the **yak** language, that may be used to develop prototypical systems and experiment with web services and behavioral types.

Keywords: Web Services, Behavioral Typechecking, Resource Usage Analysis, Type System, Type Inference, Scripting Language.

Sumário

A crescente utilização da Internet como meio de comunicação entre aplicações levou ao desenvolvimento de tecnologias para a disponibilização, descrição e comunicação entre serviços como consequência da heterogeneidade do código e das máquinas envolvidas. Deste modo, há também um grande interesse no fornecimento de "software como serviço" onde uma empresa desenvolve e disponibiliza aplicações ao seus clientes por intermédio da Internet. Os clientes dialogam com estes *Web Services* através de APIs que descrevem a interface do serviço e escondem o modo como este está alojado nos servidores. Assim, permitem criar um nível de abstracção relativamente ao fornecimento de capacidades diversas sem impor o custo de manutenção normalmente associado à sua gestão (tal como por exemplo, manutenção de hardware ou configuração e actualização das aplicações).

Contudo, as principais ferramentas disponibilizadas pelas normas e tecnologias actualmente existentes para a combinação destes serviços apenas permitem uma verificação automática limitada (baseada na comparação das assinaturas dos métodos descritos nas interfaces) e deste modo delegam a garantia de compatibilidade de comportamentos para o programador. Este fica então dependente da qualidade da documentação e da disponibilidade de tempo necessário para garantir manualmente (e sem garantias formais) a adequação da sua solução.

Nesta tese propomos um sistema de tipos comportamental, no contexto de **yak**, um protótipo de uma linguagem de scripting para *Web Services* que complementa a verificação de tipos tradicional ao garantir estaticamente a correcta utilização de serviços (tanto remotos como locais). Esta linguagem permite a descrição do comportamento por intermédio de anotações adicionais, sob uma forma semelhante a expressões regulares, que são traduzidas em autómatos finitos deterministas para a verificação. O objectivo deste trabalho é facilitar a disponibilização e criação de *Web Services* (com informação comportamental) através da incorporação de mecanismos na própria linguagem que tornam estes elementos transparentes ao programador e de utilização imediata. Disponibilizamos ainda a implementação completa do interpretador, sistema de tipos comportamental e sistema de execução para a linguagem **yak**, suficientes para o desenvolvimento de sistemas experimentais com *Web Services* e tipos comportamentais.

Palavras-chave: Web Services, Tipificação Comportamental, Análise de Utilização de Recursos, Sistema de Tipos, Inferência de Tipos, Linguagem de Scripting.

Contents

1

Introduction

1.1 Motivation

As applications grow in complexity and size so does the need for more advanced and helpful tools to assist the programmer in the development of software. As such, one of the many concerns of computer science is to provide techniques to easily and quickly verify the correction of the created code.

However, there are important limitations to the type of properties that can be extracted from a given program without falling into undecidable situations (cf. the halting problem). To tackle this limitation it is usually relied on approximations of the actual solution which are generally helpful enough, even if sometimes they can only provide conservative results. As a result, the concept of typeful programming [1] emerges, which bases itself on the notion of properties that can be statically verified.

Consequently, a typechecker is a system compatible with this methodology which includes rules to verify the abstraction, polymorphism, subtyping and modularity of a program. However, although their usefulness to program development is unquestionable, the limitations of modern type systems are becoming ever more apparent with the increase in complexity and dynamic combination of software components (for instance, with the increasing use of concurrency) as well as code reuse from different sources (that some times are only accessible through remote interfaces).

The concept of Web Services is based on the idea of interoperable communication between machines over a network. As a consequence of the growing interest from the industry, the most relevant aspects of this topic are subject to various specifications and standards, that cover different topics such as data format or even interface description/specification. This normalization eases the exchange of information between entities. However, the limitations described above are only made more clear since the interfaces are mostly related to abstract types and leave the description of the correct use of services to the documentation. In other words, this means that it is then in charge of the programmer to obey them and as such it poses inevitable scalability issues.

There is no single commonly accepted point of view to this problem. This situation leads to the existence of several different research choices for checking a type's behavior. One approach imposes conditions to each function / procedure that must be proved correct (i.e. pre/post conditions), there are already some more practical applications of this technique (for example ESC/Java [2]). Another approach focus on the ideia of combining an automaton with a type [3] which will then serve as basis for behavioral validation.

In this work we build on the latter ("state-machine") approach, centered on the verification of more standard types combined with flow restrictions imposed by a deterministic automaton. The concept of automata is well-known, and benefits from solid theoretical foundations that provide a broad set of known properties as well as efficient algorithms to handle such structures. Therefore, this topic is thoroughly explored in many textbooks covering the subject [4]. As a result, automata are widely used in a diverse set of scenarios, namely (related to the context of this work) in the verification of regular expressions, which will serve as the basis for our annotation syntax for describing the allowed behavior of a service usage protocol.

We incorporate our type system in a Java like scripting language for Web Services. Instead of focusing in the efficiency of the code, scripting languages have become a relevant programming concept for allowing to quickly and easily test new ideas with minimal coding and also by drastically reducing the hassle of getting the code to run. This languages usually rely on a small interpreter which runs the scripts instead of compiling the code directly to machine code. As such, these languages can provide an abstraction from the hardware on which they will run. Consequently, they can also offer more complex language constructions or features with the intent of easing the rapid development of applications (like automatic memory management, etc). Currently, their use covers a broad number of use cases from code execution in a web browser (JavaScript), automatizing of system administration tasks (UNIX shell), game programming (Lua, UnrealScript), construction of graphical user interfaces (Python) and many other situations. Due to the large number and quality of available libraries for building languages and interpreters, this kind of proof-of-concept prototyping technique is generally preferred as it eases the experimentation of new features without requiring too much programming work.

In conclusion, there is currently the need of designing and studying mechanisms for automatic verification of the correct use and combination of services, as a complementary correctness certification to the currently widely used type systems. Hence, the adoption of behavioral specifications at the programming language level can very positively impact the expressiveness and robustness of emerging technological contexts such as the one of Web Services, which is the main focus of this thesis, without demanding much more effort from the common programmer.

1.2 Context and description

The core of the problem is to decide before a program is run (i.e. statically) if it obeys the declared behavior, more specifically if its behavioral types are correctly used in the application code. Therefore, we intended not only to verify (even if in a somewhat conservative way) the correct flow of calls but also assure the completeness of an object's behavior[1]. This implies that trapped errors (like zero divisions, etc) are beyond the scope of this problem and as such will remain as possible sources of inconsistencies.

Without this kind of check there is always the possibility of some less obvious errors to be present in the code. For instance, situations where opened files/sockets are not safely closed after use, that could lead to loss of data or even unexpected sequences in the communication between two peers causing abnormal behavior from either party.

This problem can be approached from several different points of view. In the context of this work, we are interested in taking the Web Services approach by looking at the sequences of requests not only from the method invocation perspective but also as a service invocation. Therefore, the behavior declaration is also the protocol to which the service is expected to obey. Thus, our problem is broad enough to include the verification of the correct combination of different services as well as single objects.

As a result of this, we are required to assure the following situations:

- termination in the use of a protocol/behavior.
- reconcile the behavior of possibly different execution branches (if-else) in the most flexible way.
- correction of the behavior during and after a loop (while/repeat).
- safely interchange of behavioral values both in a method return and also as an argument to a method call, with specific implications to the termination of a protocol/behavior.
- check the consistency of a class body with the declared usage protocol, that includes the possibility of recursive (internal) methods and the storing of behavioral values.
- expand normal subtyping to include behavior, while retaining the same kind of modularity and flexibility as traditional type systems.
- correctness of behavior in the presence of exceptions.

Our type system is completely implemented into a fully functional prototype, with which prototypes of type safe distributed applications based on Web Services may be rapidly constructed, without much knowledge of details of the underlying technology.

[1] By "conservative way", we mean that our type checker may reject correct solutions, but will only accept correct ones, being the set of correct ones still large enough to allow lots of expressiveness for the programmer. This situation is expected, since in general a type system must be sound, but rarely complete.

Some other additional situations are left as possible future work (namely the handling of communication security, inheritance and concurrency) as our intention is to focus solely in the most important sequential constructions of **yak**, the language we have developed in this work.

1.3 Related work

This work includes several lines of research, using ideas from slightly different topics. For instance, the core problem of behavioral verification is to ensure a program correctly uses its resources. As such, this description includes several points of view which are reflected in somewhat distinct research choices. Therefore, it ranges from more local approaches (that look at specific aspects of programming languages) to more global views of the problem (that focus the analysis in more abstract elements like services or software components). The scope chosen for this thesis does not fit into a single topic and so ends up covering some situations in different points of that range.

In [3] Nierstrasz suggest the use of automata for this kind of behavioral verification by proposing a change in the underlying concept of type theory. He argues the abandoning of the more traditional functional (lambda calculus) based typification and instead conceptualize an object as a process rather than a function in order to handle issues of concurrency and non-uniform service availability. Although there is no formally defined type system, he no longer handles state changes indirectly and develops the idea of using automata to check the correction of his regular types. On his type framework proposal, he also expands the concept of a service as a contract which guarantees a specific behavior in regards to message exchanges during the lifetime of a type. In other words, this contract defines a protocol with the relevant information on how to correctly use the type in a way that it is possible to statically assure the code's conformance with that description. He then formalizes the principle of request substitutability and what it means to satisfy the client's expectations (request satisfiability). He also includes subtyping for his regular types in a way to ensure consistence with the principle set out by Wegner and Zdonik [5], where an instance of a subtype can always safely replace an instance of the supertype in any context it was expected. This subtyping relation assures that the subtype always understands the same requests as the supertype while also providing compatible answers. In general, this article served as an important starting point for a wide range of research on the topic by defining a more concrete development ground.

Nonetheless, the need to impose additional checks was already reflected in several previous papers but without the formal perspective suggested by Nierstrasz. For example, the Procol language [6] uses a protocol (regular expression with variables and guards) that is translated into an equivalent non deterministic finite automaton to decide on how an objects' methods can be legally used. Thus, at run-time, each object is place in a process supervised by a guard that determines its readiness to receive requests effectively shielding access to methods in accordance with the defined protocol. Although this is not a static checker, it already includes the concept of restricting the use of functions/methods to special contexts described by the protocol.

The general idea of behavioral verification split into several paths, even though they may share some common issues. Usually, the assurance of behavioral correctness revolves around the use of protocols (which are then translated into finite automata) to model the expected behavior. However, its specification method may vary with the most common descriptions being:

- directly through the listing of transitions [3, 7, 8, 9]
- through the use of syntax similar to regular expressions [10, 11, 12, 13, 6, 14]

There are still some other notations, such as graphical description of the connections in the automaton. However, these are mostly useful to ease the understanding of examples and theoretical concepts. Therefore, they usually do not appear (or at least directly) in programming languages other than in special tools designed to help the construction or specification of software architectures [15].

The choice for the notation to describe the protocol becomes rather irrelevant since it is usually possible to convert between different formats, even though sometimes this is not a trivial task. As such, the choice seems mostly related to clarity and ease of use in the particular language developed in the article. Furthermore, even when the format is similar, different solutions make use of additional notation syntax to increase the expressiveness of that particular description which means, even though the notations may resemble, there are usually small changes in the constructions of each paper's proposal.

The real use of these protocols also varies depending on when the behavioral verification actually takes place:

- run-time - where the breach of protocol raises an error [6]
- compile time - the code is statically checked to assure correctness in the use of the protocol [3, 8, 10, 11, 12]
- design time - using a more formal specification language (like Web Service Conversation Language or Architecture Description Language) that only helps at the design stage [15] or that is then fed to an automatic code generator who guarantees the correct combination of certain operations on each of the participants in accordance to the described protocol [7, 14].

It is also frequent for the solutions to end up combining more than one method of verification to provide additional safe guards that may arise from limitations in the implementation or the checking methods themselves.

Finally there are different ways to choreograph the interacting entities, in other words, it is possible to describe the interconnection of different services/components from two points of view:

- business protocols (global) [11, 16, 17] - sees the service as a whole, as a single global entity which may encapsulate several distinct elements. Thus, the different elements are seen as having a single common goal and as such the protocol aims to combine the different stakeholders into the global picture. There is even a standard developed by the W3C for a language to describe this kind of situation, the Web Services Choreography Description Language.
- end-point (local) [12, 13, 10] - only seeks to ensure compliance with the protocols on each individual types. Therefore, each entity declares its own personal protocol but leaves enough freedom on the programmer side to combine the various actors while still ensuring behavioral correctness.

In both cases the type system guarantees the correct use in accordance to the sequences allowed by the protocol. The main difference between the two lays in the chosen point of view to face the conceptual and more abstract service type that is handled by the typechecker. Both have their share of advantages and weaknesses, for example: on one hand the local view is more flexible allowing services that were developed independently to be easily combined without changing their protocols; on the other hand a global interconnection improves the documentation and clarifies how the interaction is actually carried on at dialogue layer. It is also important to notice that these two modes are not completely disjoint as they can share some results and it is even possible to map the global view into the local one [18].

Since our work is heavily inspired by [10], our type system is based on regular expression like protocol descriptions that are used to statically check (at compile time) the code correctness. We also use the more local approach by considering each object as a possible end-point of the global behavioral system. Nonetheless, our proposal also includes many deviations from that article, for instance, we also check for behavioral correctness at run-time (although this is done only for debugging reasons and not actually required by the type system).

As a consequence of these many different variations of this problem, it is difficult to identify related work without excluding some that, while not exactly in the same line as ours, include relevant contributions to the more broader topic. Therefore, we first introduce more closely related work and afterwards briefly mention some other relevant articles.

In [18] Carbone, Honda and Yoshida relate the two different paradigms for behavioral communication description (global and local) using a formal calculi based on session types. They manage to map these different perspectives while preserving the type structures by exploring a theory of end-point projection. On a later paper [19], they create a type system that allows for structured exceptions based on session types which guarantees the coordination on exception handling among communicating peers. This kind of exception types have some resemblance to ours since they also account for changes on the expected communication flow when an abnormal situations is raised. However, session types take a more business protocol point of view while our approach is more end-point related.

In following work Hu, Yoshida and Honda [20] present an implementation of a language and run-time for session-based distributed programming with additional features like session delegation and subtyping. In spite of using Java as the host language, their approach requires additional session specific syntax (besides the expected protocol notations). Also, their use of exceptions does not seem to fit in the same category as our behavioral exceptions since it looks more of a communication layer error handling than actual protocol level behavioral control. Nonetheless, they expand many interesting features related to session based programming with minimal run-time overhead.

Ravara, Vallecillo and Vasconcelos [11] propose an extension to the description of software component interfaces to include behavioral information through the use of session types. Therefore, they describe how a pairwise complex interaction between components can be provided in a high level specification. They also include a decidable proof system for this proposal.

The use of session types has some interesting possibilities as they allow for reduced complexity when checking communication compatibility, even though this will result in some loss of flexibility as a consequence of their pairwise composition. In the view of the fact that each element in the pair contains the inverse behavioral definition, the need to explicitly describe this can become a burden for more (local) object uses and as such these types are better fit for more global behavioral compositions.

Both Laneve and Padovani [17] as well as Castagna, et al. [16] follow a similar path but in a more close relation to Web Services contracts. As such, they define a specification language for services contracts and some advanced search mechanisms. They formalize the relation of services compatibility and the correct replacement of services in accordance to the provided protocols. The last paper develops an interesting use of filters as a basis for the sub-contract relation, which allows for a greater degree of freedom in the querying for services compatible with a specific protocol.

These two articles are very close to what we define for subtyping our behavioral types, even though we model internal choices directly as explicit exceptions instead of considering them as a more "peaceful" situation that may occur during the dialogue between the services. Nonetheless, this is mostly a choice in semantics linked to our own language proposal as internally they look (and behave) very similarly.

Kobayashi and Igarashi [12] take a solely local approach to what they call the resource usage analysis problem. They create a behavioral type system for a ML-like functional language and therefore similarities with our work are not directly visible. Also, their protocol expressiveness goes much beyond our own with some additional constructions. However, it is not quite clear if their use is just intended to help the formal proof of correctness or if it is suppose to have any practical role in a real world programming language. As most of the previously described work, they make extensive use of the traces concept, a heap which stores the relevant behavioral information up to a specific point in the code, instead of relying directly in automata. In subsequent work [13], Iwama, et al. expand that language to include exception handling in a behavioral sense similar to ours, since they accounts for changes in the correct use of a resource upon the raising of an exception (i.e. a change of behavior after an error).

The Vault programming language [21] created by DeLine and Fähndrich allows the programmer to define small resource management protocols that the compiler can statically enforce. These protocols can specify the order or conditions required to access or use a resource. Nonetheless their approach uses many and complicated annotations that cause a steep learning curve, even though this extension to the C language proved sufficiently powerful to handle their test cases. In [9, 8] they expand the type system to include objects and implement it in a modular static checker for languages that compile to the Common Language Runtime. This Fugue system, uses the Common Intermediate Language code with special annotations to verify, at compile time, some behavioral restrictions without causing changes in the executable code. These annotations are small rules attached to the methods of an object which group together to create specific usage protocols. The language itself has some resemblance to the experimental extended static checker for Java (ESC/Java [2]), that finds common programming errors in the code through a series of additional comment syntax that enables programmers to formally specify design aspect in pre and post condition form. However, the type of annotations available in Fugue go a step further by allowing the description of each node that forms an object's state machine that is a kind of verification that more closely resembles the one proposed in this dissertation. However, it actually splits the allowed protocols in two sets: one for controlling the access to resources (allocation and release of resources) and state machine protocols. With the first protocol it guarantees (in all paths in every method) that an object will not be used before its allocation or after it is released. With the second kind, the programmer can constrain the order of a method call to specific contexts allowed by the state machine. Additionally, it can also verify special conditions in the arguments allowing it to expand the check to more well defined domains (like SQL querys, etc). It also includes a set of rules for class subtyping based on a simplified version of behavioral subtypes proposed by Liskov, et al [22]. However, it does not include mechanisms for exception handling or concurrency. The relationship with this dissertation has more to do with the use of state machines for protocol validation since we do not use pre and post conditions in our type system. There are numerous variations of this kind of behavioral check with some additional work from the same authors [23] in regards to message passing.

In [24] Gay, Vasconcelos and Ravara expand a concept similar to the previous machine state based behavioral checking but also include session types. Therefore, in their object oriented language each method's availability is restricted by the use of pre and post conditions and also a session type which provides a global specification of its use. They also consider inheritance and subtyping relation for these dynamic interfaces. Although we have some similarities with this definition (as our types also implicitly change their interfaces based on the allowed behavior), they impose several flexibility limitations many of which (like self calls and behavioral completeness) we include in our type system.

Additionally, the following set of articles, although not directly related to our line of work (in the sense their approach follows a slightly different problem) present interesting alternative points of view to some of the objectives we intend to achieve.

Rumpe e Klein [15] describe an automaton model for the design phase of software engineering defining an operational semantics based on stream processing functions. The behavior is model through the use of automata (represented in diagrams) to describe the overall system, sub-systems and individual objects. With the created model, they discuss refinement rules (including inheritance) and their application for the construction of software.

Lee and Xiong in [25] use an interface automaton to add a behavioral type system for the Ptolemy II software framework. This framework supports concurrent component composition according to diverse models of computation. By extending the use of interface automata beyond the static typechecker and into run-type checking, they enable a safer dynamic component use.

Pavel, et al. [7] approach the problem from a viewpoint more related to software components composition by using a formal description based on Behavioural Interface Description Languages and explicit protocols. They develop the idea of using Symbolic Transition Systems as a basis for such interface description language. As a result, the actual component code is only accessible through a controller that protects the object from behavioral inconsistencies. Their implementation in Java is based on the use of code generators for those interfaces that will wrap the protocols and channels and effectively control the communication between the different components, forcing it to obey the specification. The use of channels allows for complete location abstraction as well as the notification of invalid requests.

In [14] Plasil et al, developed a technique to specify the behavior of a software component through a protocol similar to a regular expression. Therefore, they extend a Component Description Language (SOFA) with behavioral notations to describe precisely which order of method calls are allowed. They give special attention to the refinement of the design process so that such protocols appear in three levels of abstraction (interface, frame and architecture) and thus allowing the compliance verification to occur at design time as well as during execution.

Some of the previously mentioned articles include prototypes in their language proposals [6] or extend existing ones with their contributions [8, 7, 7, 25, 2].

The solution developed in here uses parts of several different ideas, even if modifying some concepts in accordance with our intended use. For example, to solve the problem of behavioral assignments we used concepts similar to those present in linear types but in a behavioral context. Even the proposed behavioral type system can also fit into a type and effect system if each behavioral restriction is looked at as a side effect of an object's method call.

Finally, the most closely related work was developed by Caires [10] since we follow a more practical path to the approach described in that article. This work is also the natural evolution of the type system in [26] (diploma thesis) by correcting and expanding many features as well as presenting a formal definition for the type system.

Web Services enjoy from an extensive coverage filled with standards, proposals and languages or libraries that help in the use of this concept, offering features that reduce the overall burden of creating or using them. However, the most common solutions are based on the use of

Web Services Description Language [27] for service description combined with SOAP [2] as the data exchange message protocol. Both are (currently) W3C recommendations that use a special XML scheme as basis for their format. Consequently, their enormous flexibility leads to an excessive complexity and verbosity which cause a steep learning curve for anyone using them directly. Despite this drawback, these technologies enjoy from greater acceptance by many companies (Google, Microsoft, etc) and therefore are present in the most common frameworks (Java 1.6, .NET 3.0, etc). There are however alternative methodologies, for instance Amazon Web Services also provides an alternative REST [3] interface. This style of architecture was introduced by Roy Fielding and is defined by a set of general principles particularly in relation to the simplicity, structure and organization of resources.

There is also a wide support for Web Services in a large number of scripting languages, usually using the combination of WSDL and SOAP. The Judo Language [4] is a full featured scripting language that includes (besides traditional features like concurrency and other object oriented elements) support for scripting communication to WSDL, JDBC and other services. It also includes threads as a primitive construction in the language itself with explicit synchronization mechanisms through the use of the lock/unlock primitive. Others more widely used scripting languages (like Python [5], Perl [6] and Ruby [7]) rely on external libraries instead of providing syntax specifically designed to be used for Web Services. Finally, there are languages more focused on the client side of Web Services (like JavaScript and its extension E4X [8], JavaFX, etc).

Since our work also handles interactions between services, there is some connection with the Web Services Choreography Description Language [28] developed by the W3C. WS-CDL defines a specific XML format that serves as basis for describing the detailed sequence of messages two peers may use during their conversation as seen from a more global point of view (the behavior visible from the outside). Since this is just a description language, it does not include concrete mechanisms for behavioral verification and instead is intended to serve more as a modelling tool that can later be fed to a code generator or type system that determines the correctness of the code with the described interface. The choreography description contains useful constructions to express parallel composition and exceptions as well as the usual standard constructions seen in normal regular expressions. This language follows the same design principle as WSDL and as such ultimately suffers from the same verbosity condition even for small examples of information exchange. Therefore, since we are more interested in exploring the principles of behavior verification, we will use a different and more simplistic format that better fits our needs instead of providing the extensive capabilities to express all possible choreography situations of the WS-CDL specification.

[2] Simple Object Access Protocol

[3] Representational State Transfer

[4] http://www.judoscript.com/judo.html

[5] http://www.python.org/

[6] http://www.perl.org/

[7] http://www.ruby-lang.org/en/

[8] ECMAScript for XML

In conclusion, in this dissertation we will address some of the issues concerning behavioral types in an object oriented language (with a syntax similar to the Java programming language). Our scripting language will have a static behavioral typechecker that will also be able to combine and use Web Services in accordance to their behavioral contract. In the future, this feature will not only prevent inconsistencies but also facilitate a more dynamic mix of services, as long as they provide a compatible protocol. From a services programming point of view, this kind of features are not that much explored in spite of the growing interest from the scientific and industrial communities.

1.4 Proposed solution

One of the main goal of this work is to define techniques to statically check a program behavior, using types to describe behaviours, and studying associated algorithmic techniques, based on the manipulation of some kind of finite state automata, to implement them. Behaviors are described by means of a regular expression like protocol that is then translated into a deterministic finite automaton to be used by the typechecker. Thus, we consider a behavioral type to be the composition (e.g, in a pair) of a normal type (something like a Java type) with the behavioral information (the protocol that restricts the use of the type - mostly its methods - to a set of specific contexts given by the automaton).

The proposed solution is based on the creation of a set of typing rules and algorithms that, when applied to a specially annotated program, will reason about its behavioral correctness. We also desire to minimize inevitable exclusions of programs that, even tough correct (at runtime), do not fit into the created type system restrictions. Therefore, we intend to minimize as much as possible the occurrence of false positives (in the identification of incorrect programs) by focusing not only on the rules' simplicity but also on their flexibility.

In order to do this kind of check we must consider the program's normal and exceptional flow as we intend to apply these conditions to all sequential constructions available in the created prototype language.

To get the general idea of this work, consider the program example in Figure 1.1.

In it we define a simple class $Travel$ that has three methods and a usage protocol. This protocol defines the specific sequence of calls that anyone using the object must obey in order to fulfill its behavior. Therefore, the entry point of the program (the $main$ method of the $Main$ class) creates an instance of the $Travel$ class and uses it correctly in terms of behavior since the protocol reaches a valid termination state (**stop**) at the end of the $main$ method's scope. There are many additional notations and more complex constructions (if-else branching, while loop, etc) that will be presented in detail in the following chapters but this example should be sufficient to get the general feel of the language and how our behavioral type concept works.

This system will be implemented in a prototype language (not only in terms of language constructions but also in the sense that the available framework is not in the same league as Java or other professional languages). It should, however, have a sufficiently large set of features so that it is easy to experiment with the proposed type system.

```
class Travel{
  usage flight;hotel;order

  flight(){}
  hotel(){}
  order(){}
}

class Main{
  main(){
    Travel t = new Travel(); //Travel#flight;hotel;order
    t.flight(); //Travel#hotel;order
    t.hotel(); //Travel#order
    t.order(); //Travel#stop
  }
}
```

Figure 1.1: A first example.

We also intend to provide some minimalistic code distribution facilities (based on synchronous communication). These will be focused on easy deployment of services which become available to be used through a web interface. In order to simplify their launch we will let the most important properties to be set as command line arguments and handle most of the distribution in a transparently way to the programmer. Therefore, we will have special syntax to declare remote types and variables which will only require the very essential information (the resource location) to reach a workable state. The prototype hides all remote requests so that there is very little difference on the source code regardless of the real location of an object. All objects are automatically exposed to the outside world when the program is launched and the server starts listening to requests on the given port number after the initialization. Although there are some security concerns that will not be addressed, the main idea is to ease the use of this communication layer with minimal programmer intervention which should reduce the learning curve and let him focus his attention on the more relevant code.

The decision to rely on the HTTP protocol for all inter-object communication was based on its extensive use on the Web Services community, the possibility of invoking method directly from the web browser and also its abilities to easily circumvent the most common communication obstacles (like firewalls). We also felt it was important to provide a simple XML base type as this kind of information structure is widely used on the Internet and it should be helpful for communicating with a browser.

The creation of these rules and algorithms implies some guarantees related to their temporal and spatial complexity with special attention to their use in the prototype language. However, given the additional set of restrictions to obey as well as since this is an initial experimental version of the implementation, we do not expect to obtain complexity results at the same level as the normal type systems. As such, our approach to the implementation will focus mostly on design decision for each isolated statement as the overall complexity of the type system can potentially become exponential for certain programs.

The validation of the type verification rules will forcefully be by means of strict formalization. In terms of the prototype, its construction will be checked based on a set of test files which will assure the coherence of the main practical results with the expected created restrictions. Finally, we also include a simple run-time obedience mechanism as a kind of "safety net" that will flag any error when the use of an object breaks the behavior described by its protocol (although it does not check the fulfillment of the protocol, only that the sequences of calls are valid).

1.5 Contributions

In this section, we summarize this thesis work, highlighting the main contributions. The main goal was to bridge the gap between the theoretical study of behavioral types in general, and their concrete application in a scripting language, actually usable in practice to construct distributed systems based on Web Services technology. This program revealed to include many challenging aspects, ranging from language design, definition of a type system, study of techniques to formalize it, and the definition of practical type checking algorithms, based on standard regular language theory, but overcoming many challenges related to how to map complex mechanisms such as parameter passing and exceptions. The main contributions are then the design and implementation of an expressive type system for Web Services, based on behavioral types, and the design and implementation of a scripting language for Web Services based on it.

1.5.1 Design and formalization of a behavioral type system

The main goal of this work is the definition of a behavioral type system including a formalization of the typing rules. We intend in future work to develop a full correctness proof of the typing system, in this work we preferred to validate the approach experimentally, concentrating on the algorithmic aspects of the typechecker, and of its integration in a scripting language, adequate to construct distributed applications.

Our approach is only meant to handle purely sequential code (no concurrency constructs). However, we deal with exception handling mechanisms, which have not been much explored in the literature; we develop here new techniques to flexible deal with exceptional behavior. Analysing the behavioral flow of a sequential program is split in checking the following programming language constructions:

- method availability, a behavioral type must limit the use of some methods (here called behavioral methods) to the specific context given by the type's protocol;
- behavioral termination, the protocol has to be completely satisfied in the sense that an object cannot fall out of scope with some incomplete behavior remaining;
- if-else, branching implies that each of those separate paths must be correct and the merging of behaviors (afterwards) must not cause inconsistencies;
- while, loops require the validation of a possible repetition in the code;

- exceptions, this includes both throwing and catching of exceptions which must account for the jump in the program flow from the raise point to the catch branch while controlling the behavioral correctness of all accessible variables; In particular, we propose a technique to compile the alternative behavioral flow patterns induced by exceptions into the same underlying automata, thus providing an uniform treatment of exceptions in the context of the other constructions.

- classes, the class body must respect its behavioral protocol internally by correctly using any behavioral field and externally by being independent of any possible external side effects that might cause inconsistencies. It must also account for any internal call that may occur on a method body, which includes the possibility of recursion;

- variable control, by using a kind of behavioral linearity (an object's behavioral view cannot be owned by more than one type). Thus, we control aliasing and also allow for storing and retrieving behavioral types;

- call control, an object's behavior can be completely passed to a method (which results in the lost of ownership) or instead only borrowed for the call scope (and therefore, the initial variable remains as that object owner);

- subtyping, code abstraction and modularity is an important feature of any programming language and therefore it is important to understand when a behavioral type can safely replace another.

Besides checking usual properties of object-oriented programs, such as method availability, our type system ensures that systems will comply with the declared protocols, even in a distributed setting, based on published behavioral specifications enriching what is usually available in WSDL declarations.

Although the type system was formalized using a standard type inference system, which declaratively specifies valid typing judgments, we also designed algorithmic syntax directed techniques, based on manipulation of finite automata, intended to provide a reasonably efficient implementation of our system.

1.5.2 Design of the programming language yak

We designed a prototypical language (named **yak**) with a syntax similar to the Java programming language which will serve as the base for the prototype. This language is not intended to provide a fully featured development framework, but it already features a basic set of tools needed to experiment and build appropriate examples to demonstrate the main features of our type system.

1.5.3 Implementation of a proof-of-concept prototype

An interpreter, typechecker, and run-time support system for the **yak** language was implemented, and is available for download at [29].

This prototype also includes a small server to launch the created services scripts (using Java Servlets). The interface description uses a specific XML format and the communication protocol will rely on HTTP. We use the REST methodology for the resource organization (in terms of URL structure). The validation of the prototype is based on a set of test files that check the conformance of the main aspects of the implementation in relation to the expected behavior. The examples are also available in the **yak** website [29].

1.6 Document overview

The rest of this document is structured as follow: this introduction is proceeded by a chapter with the formal presentation of the **yak** language and type system (its formal specification, typing judgments, etc.); next, a few examples highlight some of the language's possibilities in regards to the flexibility of the behavioral description; the third chapter describes the implementation of the prototype, its most important algorithms and how they guarantee behavioral correctness. The last chapter presents the conclusion and some possible future work.

There is also an appendix at the end that includes the full listing of the examples as well as some other minor language details and features.

2

The Yak Language and Type System

In this chapter we introduce the main aspects of the **yak** language and type system. In the first section the general syntax and an informal semantics is presented, followed by a section with a detailed description of the type system. Finally, this chapter ends with short remarks about the general idea of the solution.

As said in the introduction, this language is similar to a very simplified version of the Java language and as such its syntax and semantics should have a very familiar layout and execution model. Therefore, their presentation will be kept intentionally short. Also note that this is the simplified syntax that removes some redundant constructions (syntax sugar) in order to simplify the formalization of the type system. However, those additional constructions can be easily checked by simply combining the set of rules presented here.

The type system description uses a progressive approach in the sense that each rule is slowly introduced along side all (newly) needed operators. On each new definition we also give some examples to better express the general intention.

The main principle of this system is to consider a behavioral type as a pair of a Java-like type with a behavior described in a protocol. This behavior dynamically modifies the type to control the sequence of allowed method calls so that all possible run-time situations will produce valid call traces (in accordance to its regular-expression-like usage protocol).

2.1 The language: syntax and informal semantics

In this scripting language a program unit is simply a set of classes which entry point is the `main()` method of the `Main` class (if it is defined).

As usual, each class contains a group of fields and methods (see figure 2.1[1]). Any field must

[1] Note that any "..." means repetition is allowed.

have an unique name and method overload is type agnostic (no more than one method with the same name and number of arguments, even if their types are different). The constructor is simply a method without a return type and the same name as the class. Additionally, each method must always declare all exceptions it might throw.

As expected, the body of a method may contain any expression which will be evaluated when the method is called. The usual set of construction is allowed (figure 2.2): `E;E` is the sequential composition in the sense that the left expression is evaluated before the right one; variable declaration, which appends a new variable with the given name to the current run environment (it is always initiated with the default value of **`null`**); the assignment expression changes the content of a variable to the result of the evaluated expression; class instantiation (object creation) can only be made with the **`new`** expression; **`return`** causes the current method to end its evaluation and return the given expression as the result; **`throw`** throws an exception with the result of the evaluation of its expression. The block expression is used to create a new sub environment that is automatically destroyed after use. Finally, three control flow expressions: an **`if else`** which executes the **`if`** body or the **`else`** one if the given condition evaluates to true or false, respectively; a **`while`** loop that cycles the body until the condition turns false (the **`repeat`** expression is similar to the **`while`** except that the condition is always true); and a **`try catch`** which catches any exception thrown by the evaluation of the **`try`** body. There is also the usual method call construction which invokes a method with the given name and argument number in the target object. Method arguments may also add an **`owned`** annotation (described later) which allows (in the type system) for special behavior for that variable's content.

Types (figure 2.3) in this language are just a pair of the name of a class (or basic type) followed by the allowed usage pattern. This usage protocol is either the described in the class declaration or a more restrictive version of it. The usage protocol is a regular expression like construction with the normal choice, sequence and repeat syntax. Then there is also some more specific elements like **`stop`** for the empty behavior, simple recursion and behavioral exceptions (exceptions that cause a change in the object's allowed protocol). Identifiers in the protocol refer to a method's name (regardless of the number of arguments) except on behavioral exceptions where `N` is meant to be the type of the thrown exception of the method to which it is attached.

```
U ::= //program unit
     U U //units
   | C   //class definition

C ::= //classes
   class ID {
      usage P  //usage pattern

      ( T ID; )* //fields

      ( T ID( T ID , ... ) throws N , ... { E } )* //methods
   }
```

Figure 2.1: Abstract syntax for program unit and classes.

```
E ::= //expressions
   INTEGER | DOUBLE | STRING | XML | true | false | ID
|  this | null
|  E;E | ( E ) |
|  T ID  //variable declaration
|  ID = E   //assign
|  new ID( E , ... ) //class instantiation
|  return E
|  throw E
|  { E } //block
|  if( E ) E else E
|  while( E ) E
|  repeat E  //checkable "while(true) E"
|  try E catch( N ID ) E
//calls
|  this.ID( E , ... )
|  ID.ID( E , ... )
```

Figure 2.2: Expressions.

```
T ::= void | owned N#P | N#P

N ::= //names
   integer | double | string
|  boolean | xml | object | ID

P ::= //protocol
   (P) | ID
|  P* //0 or more
|  P+P //option
|  P;P //sequence
|  stop //empty
|  &ID ( P ) //recursion
|  ID[ N: P | ... ] //behavioral exception (changes behavior after throw)
```

Figure 2.3: Types and protocols.

```
class Bottle{
   usage open;(drink[Empty: recycle])*;close

   integer remaining;

   void Bottle(){
      remaining = 250;
   }

   void open(){ }

   integer drink(integer amount) throws Empty{
      remaining = remaining.subtract(amount);
      if( remaining.isLessOrEqualsThan(0) )
         throw new Empty();
      return remaining;
   }

   void close(){ }

   void recycle(){ }
   void recycle(string station){ }
}
```

Figure 2.4: Bottle example.

The example on figure 2.4 is intended to model the use of a small bottle. In order to drink its content it must be opened first and forcefully closed after use. Drinking over the available limit causes an exception to be thrown that forces the requirement of calling the recycle method. This example also shows that there are no special operators to the basic types, all operations (subtraction, etc.) are only available through normal method calls. All labels in the usage protocol refer to methods independent of their number of arguments (the use of `recycle` in the protocol refers both to `recycle/0` and `recycle/1`). The following example (figure 2.5) shows a valid use of the previously defined `Bottle` class.

```
class Main{
   void main(){
      Bottle#open;(drink[Empty: recycle])*;close bottle = new Bottle();
      bottle.open();
      try{
         integer drink = 2;
         while(drink){
            bottle.drink(50);
            drink = drink.subtract(1);
         }
      }
      catch(Empty error){
         bottle.recycle();
         return;
      }
      bottle.close();
   }
}
```

Figure 2.5: Use of the Bottle class.

2.2 SubTyping

Our definition of subtyping for these behavioral types obeys the substitutability principle. Therefore, a behavioral type may be safely replaced by any of its subtypes without causing any errors in the program.

Note, however, that we do not have any syntax or a concrete implementation of the notion of inheritance. This is why we use a type comparison based on the internal structure of a class (method's signatures and usage protocol) so that we still can have some additional flexibility.

(stopped top)

$$\frac{\Delta \vdash \diamond}{\Delta \vdash \mathit{object}\#\textbf{stop}}$$

(stopped sub top)

$$\frac{\Delta \vdash N <: \mathit{object} \quad P \xrightarrow{\textbf{stop}} \textbf{stop}}{\Delta \vdash N\#P <: \mathit{object}\#\textbf{stop}}$$

We do not have an abstract behavioral top type as there is no way to create an object that can safely replace any possible (non stopped) protocol. However, we do have a stopped top type as a normal object without behavior. Thus, as these two rules specify, any type is a subtype of this top if its internal class type is a valid subtype of object (always true, by definition) and if its behavior is stoppable (that is, if it can be safely stopped - even if it still has some other optional behavior left).

(ownership)

$$\frac{\Delta \vdash A <: B}{\Delta \vdash A^{\circ} <: B^{\circ}} \qquad \frac{\Delta \vdash A <: B}{\Delta \vdash A^{\bullet} <: B^{\bullet}}$$

The ownership flag is a disjoint relation: a type is either owned or not. Thus this condition can only be met if both have the exact same ownership value.

(reflection)

$$\frac{A \in \Delta}{\Delta \vdash A <: A}$$

(transitivity)

$$\frac{\Delta \vdash A <: B \quad \Delta \vdash B <: C}{\Delta \vdash A <: C}$$

(subsumption)

$$\frac{\Delta \vdash a : A \quad \Delta \vdash A <: B}{\Delta \vdash a : B}$$

The defined subtyping rules obey the expected relations of subsumption (explained above), transitivity (a subtype of a another type is also a subtype of all the other type's supertypes) and reflection (a type is both a subtype and supertype of itself).

(method subtype)

$$\frac{\Delta \vdash A'_i <: A_i \quad \Delta \vdash R <: R' \quad \{N_0, ..., N_n\} \subseteq \{N'_0, ..., N'_m\}}{\Delta \vdash m(A_0, ..., A_n)[N_0, ..., N_n]R <: m(A'_0, ..., A'_n)[N'_0, ..., N'_m]R'}$$

Method subtyping also follows the traditional conditions imposed by usual object-oriented type systems. Consequently, a sub-method must not throw more exceptions than its super-method, the return type of the sub-method must be a subtype of the super-method's return and the opposite relation for each of the method's arguments, as usual.

Before introducing the conditions for more specific class typification, it is important to remember that all behavioral methods will only be "visible" if the current usage protocol contains them. Therefore, it is not needed to require their existence in a subtype if the protocol does not contain that method. In other words, a behavioral method is only needed to be in the subtype if its call is allowed to occur within the protocol since otherwise it will always be hidden and never used (thus never causing inconsistent behavior). This is a direct consequence of the restrictions imposed by the protocol on a class' available methods.

All behavioral compatibility is done through means of simulating protocols. This operation is very flexible and allows for additional conditions in the amount and allowed number of choices and exceptions. Its description is in section 2.2.1 and therefore all references to it in here will be more superficial. In a nutshell, this operation is an expanded transition system that allows for a protocol to be forwarded not only by a single label, but also by another complete protocol. The resulting protocol is either **`stop`** when there is no behavior left, or contains the remaining behavior.

$$\frac{\begin{array}{c}\text{(class subtyping)}\\ A[P_A;;M_A]\in\Delta\quad B[P_B;;M_B]\in\Delta\\ \forall_{m_B}\in M_B\Rightarrow\exists_{m_A}\in M_A: m_A <: m_B\\ \forall_b\in P_B\Rightarrow(b\in P_A)\vee(b\notin U_B\wedge b\notin U_A\wedge b\notin M_A)\\ \forall_b\notin P_B\Rightarrow b\notin P_A\\ U_A\xrightarrow{U_B}\textbf{stop}\end{array}}{\Delta\vdash A\#U_A <: B\#U_B}$$

Normal subtyping requires the subtype to completely fulfill the supertype's behavior. In our case, this is translated into the requirement that the simulation operation at the subtype's protocol must be able to reach a stopped position after simulated with the supertype's protocol. This means any complete path in the supertype will also complete the behavior on the subtype and therefore (together with normal subtyping) the expectations of the supertype are thus completely fulfilled by the subtype.

For each method in the supertype there must be a compatible one in the subtype. However, for the reasons stated above, it may miss those methods that are both behavioral and not included in the future behavior protocol since they cannot legally be used. Thus, for all behavioral methods in the supertype (those belonging to the P_B) each one either belongs to the protocol of A (and is also behavioral) or, if it is missing from A's methods list then it cannot appear in the current usage of either B or A (it cannot belong to U_A nor U_B).

$$\frac{\begin{array}{c}\text{(class partial subtyping)}\\ A[P_A;;M_A]\in\Delta\quad B[P_B;;M_B]\in\Delta\\ \forall_{m_B}\in M_B\Rightarrow\exists_{m_A}\in M_A: m_A <: m_B\\ \forall_b\in P_B\Rightarrow(b\in P_A)\vee(b\notin U_B\wedge b\notin U_A\wedge b\notin M_A)\\ \forall_b\notin P_B\Rightarrow b\notin P_A\\ U_B\xrightarrow{?}Q\xrightarrow{U_A}V\xrightarrow{?}\textbf{stop}\end{array}}{\Delta\vdash A\#U_A \prec: B\#U_B}$$

Partial types are exactly like normal subtypes with the main difference residing in the protocol condition. In this case, it allows for a simply include condition in the sense that the partial subtype only has to satisfy a specially cut section of the partial supertype. Partial types are only meant to express a valid behavior contained inside some other, larger, type. This is particularly useful to account for partial uses of a type's behavior in variables, arguments of methods, etc.

```
class Q{
   usage a;b*+d*;c

   a(){ ... }
   b(){ ... }
   c(){ ... }
   d(){ ... }
}
```

```
class W{
   usage a;b;c

   a(){ ... }
   b(){ ... }
   c(){ ... }
}
```

Figure 2.6: Q and W examples. Both the defined classes have their method's bodies omitted as they are irrelevant to this section.

Considering the two examples in figure 2.6, with the previously defined sub-typing and partial-typing rules we can now conclude that the following relations are valid:

$$Q <: W \quad Q <: Q\#a; b; c \quad W\#a; b; c <: Q\#a; b; c \quad W \prec: Q \quad W\#b \prec: Q$$

and these invalid:

$$W <: Q \quad Q\#a; b; c <: Q \quad Q \prec: W\#b$$

Note that:

$$Q = Q\#a; b * +d*; c$$

$$W = W\#a; b; c$$

Considering the more real world example in chapter 3. From it, we can see that:

$$File\#close \prec: File\#openRead; read*; close$$

$$Block\#(cut + bend + weld + paint)* <: Block\#bend*; weld*; paint?$$

The second example (section 3.3) shows how partial subtypes help to generalise a container by only requiring to know a small path of the complete protocol. Therefore, all `*Order` classes are partial subtypes of `Order` that can then be stored inside a client's wishlist after filling the specific behavioral requirements. As such, the following relations are true (note that since the protocols of the `*Order` classes are quite large, they are omitted):

$$Order\#review*; buy? \prec: TravelOrder$$

$$Order\#review*; buy? \prec: FlightOrder$$

2.2.1 Simulation

The purpose of this operation is to forward the behavior by a well defined sub-behavior. It can be just a normal transition (if the sub-behavior is a simple label) or it may be a more complex operation when the forwarding protocol contains more than just the basic constructions.

It follows the simple template: $Protocol \overset{Forward}{\longrightarrow} Remaining$ where $Protocol$ is the simulating protocol and $Forward$ is the simulated one. $Remaining$ is the result of the operation with those two arguments that is, the behavior left after $Protocol$ has been used as if it were just $Forward$.

This simulation operations is the basis of most of the substitution conditions. Therefore, this operation can also be seen as having to decide when a temporarily substitution of behavior is allowed. As such this leads to the major design condition for these rules: the sub-behavior (the forward protocol) can not surprise the simulation behavior and thus it has to be coherent with him in terms of allowed freedoms of choice.

This brings some consequences to the choice and exception rules that should be mostly intuitive and are easily understood with appropriate examples.

$$a + b + c \overset{a+b}{\longrightarrow} \textbf{stop}$$

The protocol of the forwarding behavior does not need to account for all possible choices in the target behavior, this can be seen as those extra choices being ignored and thus not causing any unpredictable error. However, the opposite of this situation is not valid since additional choices in the simulated expression can not be matched by the forwarded protocol. As in the example, this means the protocol $a + b$ can be used as a replacement of $a + b + c$ by just ignoring the c branch. Obviously, the opposite is invalid.

$$a[N : b] \overset{a[N:b|M:c]}{\longrightarrow} \textbf{stop}$$

The case with exceptions is the opposite situation. The forwarding protocol can declare a larger number of thrown errors (in this case $a[N : b|M : c]$) than the simulated protocol ($a[N : b]$) and still be valid since all those extra exceptions can be safely ignored (they will never be thrown by the smaller protocol, any catching done will actually be “unnecessary” from his point of view but will not cause unexpected errors).

A very similar approach (but from a more Web Services contract related perspective) can be found in [17, 16] as previously mentioned in the related work section.

$$\frac{}{P \xrightarrow{P} \textbf{stop}} \tag{1}$$

$$\frac{}{P* \xrightarrow{\textbf{stop}} \textbf{stop}} \tag{2}$$

$$\frac{}{P* \xrightarrow{P*} P*} \tag{3}$$

$$\frac{P \xrightarrow{M} N}{P* \xrightarrow{M} N} \tag{4}$$

$$\frac{P*; P* \xrightarrow{M} N}{P* \xrightarrow{M} N} \tag{5}$$

$$\frac{P \xrightarrow{Q} N}{P + O \xrightarrow{Q} N} \tag{6}$$

$$\frac{O \xrightarrow{Q} N}{P + O \xrightarrow{Q} N} \tag{7}$$

$$\frac{P \xrightarrow{M} N}{P; Q \xrightarrow{M} N; Q} \tag{8}$$

$$\frac{P \xrightarrow{M} O \quad O \xrightarrow{T} N}{P \xrightarrow{M;T} N} \tag{9}$$

$$\frac{P \xrightarrow{M} V \quad P \xrightarrow{N} W}{P \xrightarrow{M+N} V \cap W} \tag{10}$$

$$\frac{T\{b/\&b(T)\} \xrightarrow{M} U}{\&b(T) \xrightarrow{M} U} \tag{11}$$

$$\frac{T \xrightarrow{M} \textbf{stop}}{\&b(T) \xrightarrow{\&b(M)} \textbf{stop}} \tag{12}$$

$$\frac{E_i \xrightarrow{C_i} V_i \quad id; Q \xrightarrow{id;N} W}{id[n_0 : E_0|...|n_n : E_n]; Q \xrightarrow{id[n_0:C_0|...|n_n:C_n|...];N} V_0 \cap ... \cap V_n \cap W} \tag{13}$$

1 This rule is the normal transition in a regular expression but with a more generic approach in the sense that it not only allows for a simple labeled transition but also any transition with the same behavior as the forwarded expression.

2, 3 As expected, the repetition construction can be removed without the need of any specific label (and thus not needing to appear in the forwarding behavior). Additionally, it can also consume itself as the simulated behavior while keeping the same behavior afterwards. This relates to the unlimited repetition nature of the $P*$ construction.

4, 5 The simulating behavior may be forwarded inside the star operator or may also require the unfolding of this repetition in order to build the appropriate matching expression.

6, 7 A choice in the forwarded protocol means the simulated behavior needs only to be valid in one of those branches.

8 This rule allows for the simulating protocol to be a prefix of the simulated expression. By matching the head of the expression (partially or completely if N becomes **stop**) the behavior can then continue with the normal protocol.

9 A sequence construction in the forwarding behavior requires its simulation to be split in two, one for the left and the other for the right side of the sequence. The result on the first side must be served as the starting point for the right side simulation.

10 A choice construction (in the forwarding behavior) is more complex to simulate since it requires each branch to be independently processed with the forwarded behavior. The calculation of the intersection for each simulation branch will result in an expression that is valid for all of those choices while also hiding the (now internal) decision from which it was generated. This intersection operation is meant to be the normal regular expression intersection and as such will not be described in this text (but can be found in [4]).

11 Recursion unfolding by replacing all labels with the recursive expression.

12 The recursive construction is simulated correctly if its body is completely matched by the body of the simulating expression.

13 The exception simulation rule has some similarities with the choice construction but with the choice being made by the possibly internal decision caused by a raised exception. In this case the intersection is not only with the result of each exception branch but also with the normal path (without any raised error).

Note: to use some of these rules it might be needed to do some **stop** stuffing, this is allowed and intentional (example, for matching the expression $a[I:b] \xrightarrow{a[I:b|O:p]}$ **stop** with any rule). It is also allowed for the reversed situation (**stop** unstuffing) by simply removing this identifier on the head of an expression.

Next we will present some practical examples of the use of these rules. In this section we use the previously introduced formal rules to reach the goal. In a later section, these same examples will be used to show the general idea of the implemented simulation algorithm.

Example 1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} \textbf{\textit{stop}}}\,(1)}{a^* \xrightarrow{a} \textbf{\textit{stop}}}\,(4)}{a^*;a^* \xrightarrow{a} \textbf{\textit{stop}};a^*}\,(8)}{a^*;a^* \xrightarrow{a} a^*}\,(-)}{a^* \xrightarrow{a} a^*}\,(5) \qquad \dfrac{}{a^* \xrightarrow{a^*} a^*}\,(3)}{a^* \xrightarrow{a;a^*} a^*}\,(9)$$

Example 2

$$\dfrac{\dfrac{\dfrac{}{a^* \xrightarrow{a;a^*} \textbf{\textit{stop}}}\,(\textit{previous example with rule 1 instead of 3})}{a^* + b^* \xrightarrow{a;a^*} \textbf{\textit{stop}}}\,(6) \qquad \dfrac{}{c \xrightarrow{c} \textbf{\textit{stop}}}\,(1)}{a^* + b^*;c \xrightarrow{a;a^*;c} \textbf{\textit{stop}}}\,(9)$$

Example 3

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} \boldsymbol{stop}}\,(1)}{a;c;c^* \xrightarrow{a} \boldsymbol{stop};c;c^*}\,(8)}{a;c;c^* \xrightarrow{a} c;c^*}\,(-)}{(a;c;c^*)+(b^*;c) \xrightarrow{a} c;c^*}\,(6) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{b \xrightarrow{b} \boldsymbol{stop}}\,(1)}{b^* \xrightarrow{b} \boldsymbol{stop}}\,(4)}{b^*;c \xrightarrow{b} \boldsymbol{stop};c}\,(8)}{b^*;c \xrightarrow{b} c}\,(-)}{(a;c;c^*)+(b^*;c) \xrightarrow{b} c}\,(7)}{(a;c;c^*)+(b^*;c) \xrightarrow{a+b} c}\,(10) \qquad \dfrac{}{c \xrightarrow{c} \boldsymbol{stop}}\,(1)}{(a;c;c^*)+(b^*;c) \xrightarrow{(a+b);c} \boldsymbol{stop}}\,(9)$$

Example 4

$$\dfrac{\dfrac{\dfrac{}{b \xrightarrow{b} \boldsymbol{stop}}\,(1) \qquad \dfrac{\dfrac{}{a \xrightarrow{a} \boldsymbol{stop}}\,(1)}{a;\boldsymbol{stop} \xrightarrow{a} \boldsymbol{stop}}\,(-)}{a[I:b];\boldsymbol{stop} \xrightarrow{a[I:b|O:p]} \boldsymbol{stop}}\,(13)}{a[I:b] \xrightarrow{a[I:b|O:p]} \boldsymbol{stop}}\,(-)$$

Example 5

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} \boldsymbol{stop}}\,(1)}{a;b^* \xrightarrow{a} \boldsymbol{stop};b^*}\,(8)}{a;b^* \xrightarrow{a} b^*}\,(-) \qquad \dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} \boldsymbol{stop}}\,(1)}{a;b \xrightarrow{a} \boldsymbol{stop};b}\,(8)}{a;b \xrightarrow{a} b}\,(-)}{a[I:a;b^*];b \xrightarrow{a[I:a]} b}\,(13)$$

2.3 Type System

The main purpose of any type system is to check and assure the validity of a set of properties in a program. In order to do this kind of reasoning it relies on well defined typing rules. Therefore, each construction on the program syntax has one or more rules that impose specific constraints to be checked or applied to that particular block of code. Once the type system covers all possible syntax constructions it is then possible to prove the soundness of the whole system in regards to the properties it is intended to vouch.

Each rule is meant to be the building block that forms the complete type system. However, even these rules are also themselves built using additional building blocks based on formal logic to described the explicit conditions they require. Besides normal logic, it is also frequent to use special types of judgments which have a more friendly syntax (making the description easier to read and write).

Before introducing the format of our main typing judgment it is important to first take note of the general idea of this type system. As described above, the main goal is to assure the correctness of the behavior on all used objects. This behavior is described by a protocol (that has a syntax similar to a regular expression) which is then attached to a kind of type normally used in object oriented languages. As the program flows, the allowed behavior of the objects is changed by each language construction in some specific ways. Therefore, our basic typing judgment needs to account for these side-effects by having have a special position for the resulting modifications.

(typing judgment)

$$\Delta_{before} \vdash E : T_{result} \mapsto \Delta_{after}$$

Judgment notation All judgments follow this template on which the type check of an expression E in the environment Δ_{before} results in a type T_{result} and causes side-effects on the initial environment turning it into environment Δ_{after}.

Please mind that throughout this presentations we try to keep the overall notations simple by avoiding introducing new symbols and instead rely on indexes to differentiate among possibly ambiguous elements/constructions. For example, all environments share the letter Δ but an appropriate label is used in its index should there be more than one (different) instance of it in the same rule. Also note the names used as expressions (E), types (T), etc. are intentionally referring to the respective grammar construction. Each additional symbol or operation will be introduced and described as its role becomes relevant for detailing the intricate concepts of a rule.

The main target of this type system are classes. They allow for the creation of new behavioral types and their methods are the only section in the grammar that may contain expressions. Therefore, a class is a type that includes a set of variables (fields, always private) and methods (always public). It also defines a usage protocol that restricts the availability of some of the methods to specific traces given by that behavior protocol. Since the use of any field is forcefully split among a class' methods, it is necessary to verify their behavioral correctness by checking the internal consistency of the class. For this the type system must not only verify the behavior in normal expressions, but also their behavioral context given by the possible call scenario allowed for that method as expressed by the class' usage protocol.

2.3.1 Basic constructions

We will now describe two constructions (sequence and block) since their simplicity should help to get the general idea of the type system without having to understand many of its details. From the grammar constructions it should be clear that these two elements will always be checked from within a method's body (as that is the only place they can appear). Therefore, their typing environment will already be fully built and therefore this rule only needs to focus on the effects produced to it.

(sequence)

$$\frac{\Delta \vdash E_0 : T_0 \mapsto \Delta' \quad \text{stopped}(T_0) \quad \Delta' \vdash E_1 : T_1 \mapsto \Delta''}{\Delta \vdash E_0; E_1 : T_1 \mapsto \Delta''}$$

sequence This expression starts by checking the left side and proceeds afterwards with the checking of the right one. Since this is a sequential construction any side-effects produced by the left side must be carried on as the starting point for the checking environment of the E_1 expression. However, the result of checking E_0 is not stored anywhere and will be lost after its evaluation. This situation justifies the additional requirement of the resulting type to be in a stopped state (definition 1) so it can be safely discarded without the danger of losing references to objects with incomplete behaviors. Also note that the environment after this construction is checked is the one resulting from all the right side validation, therefore all the changes are chained in a sequential order.

(block)

$$\frac{\Delta \uplus \Delta_{block} \vdash E : T \mapsto \Delta' \uplus \Delta'_{block} \quad \text{stopped}(\Delta'_{block})}{\Delta \vdash \{E\} : T \mapsto \Delta'}$$

block A block expression allows for the creation of a new sub environment with a well defined scope. Since its life is limited to the curly brackets, any variable declared inside it will automatically become unreachable after the block ends. From a behavioral point of view these soon to be destroyed objects must all be in a stopped condition in order to avoid some left over protocol. Since we do not allow for duplicated names in the environment the disjoint union (definition 2) is used to split the environment resulting from the checking of E into the one that will be lost (Δ'_{block}) and the remaining one (Δ'). Finally, note that any expression inside the block is checked with the new environment in place but only the side-effects in the initial environment are considered for the final resulting one (Δ').

Environment notation The environment (Δ) is a special structure that holds all types (declared classes) and variables that are reachable in the current scope. It also holds a checking context that will only be described later. It is not allowed to have duplicated names in the environment and therefore each name uniquely identifies a variable or type.

When it is needed to explicitly use the empty environment the symbol $\emptyset$ is used.

Variable notation Each environment variable has a static type related to the typing annotation in the source code and a dynamic type that stores the changing state during the typecheck. The static type expresses what the variable is allowed to save (write) and the dynamic part contains the current state of its content (read). Since the dynamic type may change on each call the information must be stored separately. We chose to allow for a variable to refine an objects' allowed behavior by restricting the protocol into something more specific. This means that on an assignment the allowed behavior can be shortened into some more particular case. This will only be shown on some rules further down but it is necessary to understand the why of the variable separation into static and dynamic parts.

This information obeys the syntax $x : T_{static} \times T_{dynamic}$ for a variable labeled x stored in the environment.

Example 6 (Environment notations) *Examples of environment notations containing only variables (no other previously declared types or context shown).*

$$\begin{aligned} &\emptyset\text{: empty environment.} \\ (x : T_{static} \times T_{dynamic}), (y : T_{static} \times T_{dynamic})&\text{: environment with variables labeled x and y.} \end{aligned}$$

Definition 1 (Stopped) *Any type or environment is stopped if and only if it does not have any behavior left in it. A* ***void*** *type is also accepted as stopped. The environment version of this condition is just an iteration over all variable it contains, note that the stopped condition is only applied to the dynamic type of a variable since it is the one that stores the changing behavior.*

$$\begin{aligned} stopped(T) = stopped(N\#P) &\Leftrightarrow N\#\textbf{\textit{stop}} \\ stopped(\textbf{\textit{void}}) &\Leftrightarrow \textbf{\textit{true}} \\ stopped(\Delta) &\Leftrightarrow \forall (x : T \times D) \in \Delta : stopped(D) \end{aligned}$$

Definition 2 (Environment split / concatenation) *This operations splits and environment (Δ) into a set of smaller ones (Δ_x) which are disjoint among themselves. The complementary operation takes a small set of disjoint environments (Δ_x) and merges them into one (Δ).*

$$\Delta = \Delta_0 \uplus ... \uplus \Delta_n \text{ (disjoint split)}$$

$$\Delta_0 \uplus ... \uplus \Delta_n = \Delta \text{ (disjoint concatenation)}$$

This operation can be interpreted as either a disjoint split of the environment into two (or more) different ones or a concatenation of two (or more) disjoint environments depending on the flow of the operation.

Definition 3 (Environment contains) *An environment (Δ) contains an element if that element is stored inside it.*

$T \in \Delta$: *a type is in an environment if it contains that type's declaration.*

$x \in \Delta$: *a label is in an environment if it contains a variable with that name.*

An environment may contain several elements inside it. Therefore, this operation accepts different kinds of arguments that all search over an environment.

2.3.2 Control flow

We will now present the most relevant control flow expressions that are responsible to operate the normal flow of a program (as opposed to the exceptional flow on section 2.3.5). Thus, it is needed to consider the expected order the evaluation may use so that all possible behavioral situations can be handled by these rules.

$$\frac{\Delta \vdash E^{cond} : \textbf{boolean} \mapsto \Delta_{cond} \quad \Delta_{cond} \vdash E^{if} : T \mapsto \Delta_{if} \quad \Delta_{cond} \vdash E^{else} : T \mapsto \Delta_{else}}{\Delta \vdash \textbf{if}(E^{cond})\ E^{if}\ \textbf{else}\ E^{else} : T \mapsto \Delta_{if} \sqcap \Delta_{else}}\ \text{(if else)}$$

if-else An **`if else`** expression is a fork in the code flow that has a possible merge point at its end. This split path starts after the condition (which will select the flow at run-time) and therefore both branches use its resulting environment (Δ_{cond}) as their initial state. Since this construction is also meant to be used as a conditional expression, the resulting type of both branches must be the same. However, the same restrictiveness for the environment would be rather strong and intrusive. Therefore, this rule just requires an environment that can be safely used regardless of the selected branch. This intersection operation (definition 4) merges the two environments from those different flows. This means that after this expression, the environment contains the shared behavior of both branches and the check can proceed independently of the chosen branch. (note that a **`return/throw`** in any of the branches will result in an empty environment which is a predictable argument for the intersection)

In the environment intersection operation (definition 4) we use the concept of behavioral subtype ($subtype <: supertype$) that was described in section 2.2. As a quick remainder, our behavioral subtype borrows an initial condition from normal object oriented type systems in regards to method and argument compatibility. However, this condition is then extended to add the behavior of types into the equation. It also remains faithful to the concept of allowing any code using a specific type to be safely replaceable by any of its subtypes without causing any unexpected errors. Thus, a subtype must always have a compatible behavior with the super-type (it has to allow itself to be used with the supertype protocol) even if it also allows for some additional freedom of use. A very simple example is $T\#a + b <: T\#a$, note that the reversed relation is invalid.

Definition 4 (Environment intersection) *The intersection of environment Δ_A and Δ_B contains each variable (x) contained in both of those environments. However, while the variable keeps the same static type (T), its dynamic type (D) must be a valid subtype of the variable's dynamic type on both environment Δ_A (D_A) and Δ_B (D_B).*

$$\begin{aligned}
\emptyset \sqcap \emptyset =& \emptyset \\
\Delta \sqcap \emptyset =& \Delta \qquad \emptyset \sqcap \Delta = \Delta \\
\Delta_A \sqcap \Delta_B =& \{(x : T \times D) : (x : T \times D_A) \in \Delta_A \wedge (x : T \times D_B) \in \Delta_B \wedge (D <: D_A) \wedge (D <: D_B)\}
\end{aligned}$$

This operation gives the common "future" shared by both environments. This implies that for each common variable its dynamic type must either be in an accept state or that there exists at least a single common path to reach one (or more) accept states. This is more easily expressed as the resulting dynamic type being a subtype of the two intersecting terms. The empty environment is ignored by this operation for having no influence on the result. With the exception of the empty environment, this operation is only valid if both environments share the same number and name of declared variables and if each pair of similarly named variables has the same static type.

In our case the subtype relation is used to extract the minimal behavior shared by both dynamic types as the subtype condition effectively trims all non shared behavior between D_A and D_B. In algorithmic form, this shared path can be easily obtained by simply intersecting the two protocol expressions.

Example 7 (Environment intersection)

$$\begin{gathered}
(x : T_x \times A\#a; a; a), (y : T_y \times B\#a + b + c), (z : T_z \times C\#((a; c) + (b; c*)); (w * +y)) \\
\sqcap \\
(x : T_x \times A\#a*), (y : T_y \times B\#c + d + e), (z : T_z \times C\#(a + b); c*; w) \\
= \\
(x : T_x \times A\#a; a; a), (y : T_y \times B\#c), (z : T_z \times C\#((a; c) + (b; c*)); w)
\end{gathered}$$

Example 8 (Invalid environment intersection)

$$(x : T_x \times A\#a; a; a) \sqcap (x : T_x \times A\#\textbf{\textit{stop}}) = \textbf{\textit{ERROR}}$$

Our **`while`** and **`repeat`** expressions share a common base with the main difference residing in the loop condition. The **`while`** loop tests on each iteration the condition to determine if it should continue or not. On the other hand, the **`repeat`** expression will never stop (and thus, at run-time, is equivalent to a **`while`** loop with an always true condition). The necessity for this (apparently) redundant repeat construction arises from the need to statically know the probability of encountering this forever-loop without breaking the modularity and abstraction of the typing rules. Its practical use will only be made clear on a later example which makes use of recursion on exceptions (see section 3.1 method `loopOpenRead`).

$$\text{(while)}\quad \frac{\Delta <: \Delta_w \quad \Delta_w \vdash E^{cond} : \textbf{boolean} \mapsto \Delta_{cond} \quad \Delta_{cond} \vdash E : T \mapsto \Delta_w \quad \text{stopped}(T)}{\Delta \vdash \textbf{while}(E^{cond})\ E : \textbf{void} \mapsto \Delta_{cond}}$$

while This construction allows for a cyclic evaluation of the condition and body until the condition no longer holds true. Therefore, it follows a sequence of evaluations that always ends after that condition. For this reason, the environment after a **`while`** loop is the result from the side-effects produced by the check of E^{cond}.

The resulting type from the condition must be of boolean nature so that a choice can always be made. Note that this is one of the basic types and as such it will never leave any incomplete behavior (it is always equivalent to **boolean#stop**). However, the resulting type from the evaluation of the body (T) might be more complex so, before the type falls out of scope, it is required for it to be in a safe state (i.e. stopped, as defined above).

The **`while`** loop can have more side-effects than just the one caused by the evaluation of the condition. These cases can occur when there is some choice done in the body of the **`while`** that influences the allowed behavior after this expression (a case similar to using the **`while`** as a pseudo **`if`**). In order to account for this, the environment subtype condition (definition 5) is used to select an environment (Δ_w) that is both the one with the largest behavioral freedom and yet still accepts the body of the **`while`**. Note that this filter is done before the check of the condition or the body and thus the environment before the condition and after the body are the same (which will allow for the loop to continue).

The situations covered by this rule are further described in the implementation section with actual examples (see section 4.3.2).

Definition 5 (Environment subtyping) *An environment (Δ_A) is a subtype of another environment (Δ_B) if for each commonly labeled variable (x) its dynamic type (D_A) is a subtype of the dynamic type in the supertype environment (D_B).*

$$\Delta_A <: \Delta_B \quad \Leftrightarrow \forall x : (x : T \times D_A) \in \Delta_A \wedge (x : T \times D_B) \in \Delta_B \Rightarrow D_A <: D_B$$

Example 9 (Environment subtyping)

$$(x : T_x \times A\#a*), (y : T_y \times B\#a + b + c), (z : T_z \times C\#c[N : b + c|E : a*])$$
$$<:$$
$$(x : T_x \times A\#a; a; a), (y : T_y \times B\#a), (z : T_z \times C\#c[N : b|E : \textbf{\textit{stop}}|R : a; a])$$

Note the inverse relation is invalid.

$$\text{(repeat)}\quad \frac{\Delta \vdash E : T \mapsto \Delta \quad \text{stopped}(T)}{\Delta \vdash \textbf{repeat}\ E : \textbf{void} \mapsto \emptyset}$$

repeat This expression is in every sense equal to a **`while`** with an always true condition but without requiring the static checker to look "deep" into the resulting type of the condition. Also, since there is no syntax to abort this loop, the environment after it is always empty as the code afterwards is unreachable.

behavioral linearity So far the rules created assume there is no interference in the behavior of each expression evaluation. However, to cause this restriction it is necessary to use a kind of behavioral linearity. This condition is modeled by an ownership flag that is intended to express if a type has an exclusive view of the object's behavior in the sense that no one else can know its current allowed behavior. Therefore, an owned type has special permissions to store and control the object. Note, that for the stopped protocol (#**stop**) its value is irrelevant since there is no behavior to cause interferences. As such, in this case, the flag is only shown if there is some other important side-effect.

Type notation The two basic versions of the type notation (short (T) and full ($N\#P$)) can have an additional ownership specifier: T° for an owned type or $T^{\bullet}$ for a non-owned. When the owned flag is not explicitly give (like in T) it means any owned value (owned or non-owned) is allowed and it will not be changed in that rule. Thus, the type keeps its ownership value even if its current behavior may change. The owned flag is always explicitly expressed with one of the two previous annotations if the value is required to change or have a concrete value. It is s also possible to assign the ownership value to a label with the notation T^{s} (ownership value is labeled s).

$$\text{(return)}$$
$$\frac{\begin{array}{c}\Delta \vdash E : T^{\circ} \mapsto \Delta' \uplus \Delta'_{fields} \quad \text{stopped}(\Delta') \quad \langle \Omega; \Theta \rangle \in \Delta \\ (... \ll Method(...)[...]T_{return} \gg \Delta_{fields-after}) \in \Omega \\ \Delta_{fields-after} \lhd \Delta'_{fields} \quad T <: T_{return}\end{array}}{\Delta \vdash \textbf{return}\ E : \textbf{void} \mapsto \emptyset}$$

return The **`return`** expression requires the knowledge of two contextual information which is related to the current method under check: the return-type and the fields state after the method.

The first is needed to check if the returned-type is a valid sub-type of the method return-type, as also required by any other type systems with methods/functions. The possession of a type must be always completely transferred on a **`return`** expression. The need to require this ownership arises from the fact that the return-type can have some (non stopped) behavior. However, the method call construction does not force the immediate use of this behavior and therefore it could be interleaved with other calls which could cause unpredictable problems (due to type abstraction) if some internal field is just "borrowed" to the outside of a class (instead of completely transferred).

This **`return`** construction causes the termination of the current method execution by returning the evaluated expression. All local (stack) variables will become out of scope as a consequence of this action. Therefore, all of them must be in a stopped state so no incomplete behavior is left behind (note this must also be true to the method's arguments).

Also note that the environment becomes empty after the check of this expression since the flow of the program stops.

Finally, the local fields of the class must be in a consistent state with the environment after the call, in other words, the behavior of all fields must be compatible (intersection-wise) in every possible **`return`** so that the state of the class after a call is always coherent (i.e. it does not diverge to different paths in the same method - this kind of behavior is avoided since it could cause results that are not statically enforceable by requiring the behavior to be dependent on a specific **`return`** expression within the method). This is done by using the reversed intersection (definition 6) of the method end environment (the state of the fields after the method ends) with the environment after the **`return`** expression is checked.

It is time to introduce the environment's contextual information which stores the previously mentioned environment's state at the end of the method. This is a consequence of all checks only happening inside the class consistency check (explain further down) and therefore it is always possible to know which class and which method the verification is currently testing.

Context notation (methods) As implied by the return rule, one of the information stored in this structure is the state of the environment on each method call. It contains the behavioral situation of each field variable both after the call ends and also before it starts. This map of methods is store inside Ω on the contextual information ($\langle\Omega;\Theta\rangle$) of an environment and follows the template: $\Delta_{before} \ll Method \gg \Delta_{after}$. The other structure in the environment (Θ) is related to the exceptions and will only be presented in that section.

Method notation The template $m(T_{arg_0}\ \ n_0, ...)[T_{excp_0}, ...]T_{return}\{E_{body}\}$ is used for all methods: m is the name of the method and the exceptions list (inside []) can be omitted if the method does not throw any exception. Actually, any omission in a complex structure is equivalent to a "do not care" value in order to simplify the rule's notation.

Definition 6 (Environment reversed intersection) *The reversed intersection of environments expresses that Δ_A is the result of intersecting environment Δ_B with some other environment $\Delta_{unknown}$.*

$$\Delta_A \lhd \Delta_B \ \stackrel{\triangle}{=}\ \Delta_A = \Delta_B \sqcap \Delta_{unknown}$$

The previously (valid) environment intersection example can also be applied to this case by simply rearranging the terms to the appropriate positions.

2.3.3 Variables

Variables are the crucial structure for tracking behavior in our type system. Consequently, all behavior changing actions will have effects in the variables state. Remember that all variables have two internal types: one static - the source code type annotation; and the other dynamic - the previously mentioned changing behavior. This kind of change is what causes the environment to suffer possible side-effects on each check.

These rules require a basic notion of partial subtyping ($subtype \prec: supertype$). This relation is similar to normal subtyping with some differences on the behavioral side, namely a partial subtype is only require to partially satisfy the behavior of its supertype. This means a partial subtype behavior is contained somewhere inside the behavior of the supertype allowing a safe (but only temporarily) use of it. A simple example: $N\#b \prec: N\#a;b;c$ (the inverse is invalid).

$$\text{(variable declaration)}\quad \frac{x \notin dom(\Delta) \quad T_{class} \in \Delta \quad T \prec: T_{class}}{\Delta \vdash T \quad x : \textbf{void} \mapsto \Delta \uplus (x : T^{\circ} \times \mathsf{stop}(T))}$$

variable declaration A variable declaration simply appends a new variable structure to the environment with the given label. As usual, it is required for the label to be unique in the environment (no variable x must be inside the domain of the initial environment). Note the static type has the owned flag set as all declared variables always require the ownership of type (this does not include method's parameters as their declaration can allow for non owned arguments). The initial content of a stack variable is stopped since there is no real content in it yet.

Definition 7 (Stop type/environment) *Any stop structure is either **void** or has a **stop** pattern.*

$$\begin{aligned} stop(T) =& stop(N\#P) = N\#\boldsymbol{stop} \\ stop(\boldsymbol{void}) =& \boldsymbol{void} \\ stop(\Delta) =& \{(x : T \times stop(D)) : (x : T \times D) \in \Delta\} \end{aligned}$$

This operation just filters out any behavior that might be inside the type or environment to which it is applied to.

We do not differentiate among stack, field or argument variables as their typing rules are all the same. However, their creation and use has special conditions which are presented in further sections. For now it is just important to know that non owned variables are basically constant (as they cannot receive any new content) and field variables have their behavior split among possibly several class methods.

$$\text{(assignment)}\quad \frac{\Delta \vdash E : T^{\circ} \mapsto \Delta' \quad \Delta' = \Delta'' \uplus (x : T^{\circ}_{static} \times T_{dynamic}) \quad \mathsf{stopped}(T_{dynamic}) \quad T <: T_{static}}{\Delta \vdash x = E : \textbf{void} \mapsto \Delta'' \uplus (x : T^{\circ}_{static} \times T^{\circ}_{static})}$$

assignment To avoid some conflicts in the value of the ownership flag, all variables of a non owned type are read only (constants). This is intended to removed situations where the ownership flag might change in different executions paths (like in an **`if else`**). Thus, the assignment rule requires an owned type as the static type of the variable. Because of this, and since an owned type expresses the absolute possession of that object, before an assignment can take place it is required the previous content to be stopped since its ownership will be lost.

The subtype relation is used to assure the complete ownership can move to this variable without some incomplete behavior being left behind. Also note that since the previous test is valid, the type can now be safely exchanged to the more restrictive version declared in the static type of the variable.

$$\text{(variable owned read)}\quad \frac{\Delta = \Delta' \uplus (x : T^{\circ}_{static} \times T^{\circ}_{dynamic})}{\Delta \vdash x : T^{\circ}_{dynamic} \mapsto \Delta' \uplus (x : T^{\circ}_{static} \times \mathsf{stop}(T_{dynamic}))}$$

variable owned read Behavioral linearity implies the ownership value must be uniquely kept by a variable. As a result, an owned read must force the old owner to completely lose the possession of that type. This is caused by stopping the dynamic type it contained while returning the owned view of the same type to the reader.

$$\text{(variable non-owned read)}\quad \frac{\Delta = \Delta' \uplus (x : T_{static} \times T_{dynamic}) \quad T_{dynamic} = N\#V \quad V \xrightarrow{W} Q}{\Delta \vdash x : N\#W^{\bullet} \mapsto \Delta' \uplus (x : T_{static} \times N\#Q)}$$

variable non-owned read Although we require a sort of linearity for the ownership of a type this can be somewhat circumvented by non-owned read. This read models a kind of borrowed ownership where the use of a type is strictly limited to well defined protocol. With this rule, a type can be passed as an argument to a method without becoming stopped as long as its state after the call is predictable (and its ownership is not required). In other words, if the method uses the object in such a way that its protocol can be forwarded to another valid point then the type can still be used correctly afterwards.

The predictability of the type's sub-use is calculated by simulating the non-owned use (W) on the complete behavior (V). This operation (with the format $Protocol \xrightarrow{Forward} Remaining$) is completely defined in the section 2.2.1.

2.3.4 Calls

Method calls will cause a transition in the current behavior of a object. As explained in the previous section, only variables control the allowed behavior and therefore method calls can only occur on objects stored inside these structures. By requiring this we also avoid numerous situations where new instances or returned values might be lost without having completed their behavior. As such, this condition will force all values to be stored and thus eliminating the danger of lost references to incomplete (with some behavior left) objects.

However, not all calls will cause a modification of behavior. Since not all class' methods may appear in the usage protocol, we decided to allow those method to be freely used. This means that a protocol will only restrict the use of a subset of all class' methods. We hereby call behavioral methods those whose name is contained in the initial usage protocol on the class declaration and thus can only be called in specific allowed contexts (described by the class' protocol). This can only happen if the non behavioral calls do not cause any change in the class fields state. This will have more clear implications on the consistency check described in section 2.3.8.

All following rules share some common aspects, namely: all require for the method to exist in the class' method list; behavioral methods must belong to the class' protocol (expressed by $m \in Protocol$) except for non-behavioral ones ($m \notin Protocol$). The other shared condition, is related to the evaluation of the arguments in a call.

call argument check In order to avoid possible interference in the evaluation of each argument it is required the disjoint split on the evaluation of each one of them. Although the ownership flag correctly avoids cases where the duplicated use of the same object could cause interferences, it is insufficient to stop errors on non-owned reads. In this case, since the non-owned reads explicitly (only temporarily) break behavioral linearity for the borrowing of a section of behavior, it is necessary to have additional conditions to avoid the same kind of interferences that might occur if we did not have the owned flag. Therefore, it is required for all arguments to be environmentally disjoint in the sense that each resulting argument object cannot be overlapping with a previously calculated one. If this were not the case some non owned types could be passed as two distinct arguments and the check of the method's body would not be able to detect it as it assumes all arguments have disjoint behaviors (i.e., there is no relation between the behavior of two arguments).

Before stating the call rules it is just necessary to introduce the class' notation.

Class notation The format of the structure for the class' content is $Name[Usage; Fields; Methods]$. The $Usage$ contains the behavior (protocol) of the class, $Fields$ contains a map of the class' fields, $Methods$ contains the methods and the constructors.

(behavioral method call)

$$\frac{\begin{array}{c} T \in \Delta \quad T = N[Protocol; ...; Methods] \quad m \in Protocol \quad m(T_0^{s0}, ..., T_n^{s_n})[...]T_r \in Methods \\ \Delta = \Delta_{E_0} \uplus ... \uplus \Delta_{E_n} \quad \Delta_{E_i} \vdash E_i : T_{arg_i}^{s_i} \mapsto \Delta'_{E_i} \quad T_{arg_i} <: T_i \\ \Delta'_{E_0} \uplus ... \uplus \Delta'_{E_n} = \Delta' \quad \Delta' = \Delta'' \uplus (x : T_{static} \times T_{dynamic}) \\ T_{dynamic} = N\#M \quad M \xrightarrow{m} O \end{array}}{\Delta \vdash x.m(E_0, ..., E_n) : T_r^{\circ} \mapsto \Delta'' \uplus (x : T_{static} \times N\#O)}$$

behavioral call The only remaining conditions to be described are related to the simulation (or forwarding) of the protocol. After a method has been called the variable's dynamic state must transition to the next allowed behavior. This is done with the same simulation operation (see section 2.2.1) as before but with only one simple label (the name of the called method).

$$\text{(non-behavioral method call)}$$
$$\frac{\begin{array}{c} T \in \Delta \quad T = N[Protocol; ...; Methods] \quad m \notin Protocol \quad m(T_0^{s_0}, ..., T_n^{s_n})[...]T_r \in Methods \\ \Delta = \Delta_{E_0} \uplus ... \uplus \Delta_{E_n} \quad \Delta_{E_i} \vdash E_i : T_{arg_i}^{s_i} \mapsto \Delta'_{E_i} \quad T_{arg_i} <: T_i \\ \Delta'_{E_0} \uplus ... \uplus \Delta'_{E_n} = \Delta' \quad (x : T_{static} \times T_{dynamic}) \in \Delta' \\ T_{dynamic} = N\#M \end{array}}{\Delta \vdash x.m(E_0, ..., E_n) : T_r^{\circ} \mapsto \Delta'}$$

non-behavioral call Non behavioral calls have no side-effects and therefore this rule is mostly just checking for argument correction and that the method with the specific number of arguments exists inside the type.

$$\text{(new instance)}$$
$$\frac{\begin{array}{c} T \in \Delta \quad T = N[Protocol; ; Methods] \quad Class(T_0^{s_0}, ..., T_n^{s_n})[]\textbf{void} \in Methods \\ \Delta = \Delta_{E_0} \uplus ... \uplus \Delta_{E_n} \quad \Delta_{E_i} \vdash E_i : T_{arg_i}^{s_i} \mapsto \Delta_{E_{i+1}} \quad T_{arg_i} <: T_i \\ \Delta'_{E_0} \uplus ... \uplus \Delta'_{E_n} = \Delta' \end{array}}{\Delta \vdash \textbf{new} \;\; N(E_0, ..., E_n) : N\#Protocol^{\circ} \mapsto \Delta'}$$

new instance A new instance simply creates a new type with the behavior protocol of the class' usage declaration. This type must be owned since the newly created type cannot possibly be held by someone else.

The only remaining type of call is the internal one. We decided to allow for any method to be able to call all other remaining internal methods. Thus in our type system, the behavior is only intended to restrict the use of a type outside the call context. As such, some code using the type must follow the specific sequence of calls that the behavior describes. However, for the internal code of the class this is not required. Nonetheless, special considerations must be taken for this kind of behavioral recursion as the state of the fields must be carefully managed when a method calls itself or for when some other kind of internal loop may happen.

$$\text{(internal call)}$$
$$\frac{\begin{array}{c} T \in \Delta \quad T = N[Protocol; ; Methods] \quad m \in Protocol \quad m(T_0^{s_0}, ..., T_n^{s_n})[...]T_r \in Methods \\ \Delta = \Delta_{E_0} \uplus ... \uplus \Delta_{E_n} \quad \Delta_{E_i} \vdash E_i : T_{arg_i}^{s_i} \mapsto \Delta'_{E_i} \quad T_{arg_i} <: T_i \\ \Delta'_{E_0} \uplus ... \uplus \Delta'_{E_n} = \Delta' \\ \langle \Omega; \Theta \rangle \in \Delta' \quad (\Delta_{fields-before} \ll m \gg \Delta_{fields-after}) \in \Omega \\ \Delta' = \Delta'' \uplus \Delta_{fields-before} \quad (\textbf{this} : T_{static} \times T_{dynamic}) \in \Delta' \quad T_{dynamic} = N\#M \end{array}}{\Delta \vdash \textbf{this}.m(E_0, ..., E_n) : T_r^{\circ} \mapsto \Delta'' \uplus \Delta_{fields-after}}$$

internal call As said in the previous section, we use a special structure (Ω) to store the state of the class' fields before and after all internal calls and so this rule is much simplified as it is only needed to first check compatibility with the before environment and then change to the after state. Note, however, that this also has some implicit effects namely on the non behavioral methods. Since they can not change the state of the fields, any call to a behavioral method from within them is only permitted if that same methods will not modified the internal fields. Therefore, this kind of internal recursion can be seen as pushing the code from some other method to the body of the current one.

2.3.5 Exceptions

Our run-time approach to exceptions is in every sense similar to the usual (Java) way. Therefore, a normal program flow may be abruptly stopped and flag an error carried on by the raised value. Any method that does not catch this value will be cut through until an appropriate **`catch`** expression is reached. In our case the catching is done only by the name of the error type. This type can not have any behavior left (it must be stopped). By keeping this mechanism behavior agnostic the run-time will not need to track each protocol which also makes the catching more simple (since it will not need to compare behaviors). Undeclared exceptions could lead to unpredictable termination points inside the code and thus are incompatible with our type system. The kind of expected exceptions are stored in the remaining (and not yet introduced) structure (Θ) inside the contextual information of the environment.

Besides requiring for each exception to be (eventually) caught, so to avoid abnormal terminations of a program that might leave some dangling behavior, our exception system also requires for each exception to be behavioral. A behavioral exception causes the internal protocol of an object to change after it throws some other value. This is intended to model some special behavior that only is used for the handling of a special situation.

Our protocol syntax has probably the best example to better express this idea. Consider the protocol $method[ThrowType : special; done]; done$, this means a method named $method$ is allowed to **`throw`** an exception of type $ThrowType$ that will change the objects' allowed protocol to $special; done$ instead of just $done$ of a normal flow (when no exception is raised).

Also note that although all our exceptions are behavioral, normal exceptions can be easily translated to this by simply assigning the old behavior as the target of the change (and thus causing no real behavioral modification).

Context format (exception handlers) The exception handlers are kept inside the right context container labeled Θ (for an environment context with the format $\langle\Omega; \Theta\rangle$). This map stores, for each expected exception, two environments: the top catch environment ($\Delta_{try-catch}$, the environment on which the exception is caught - in other word, the environment where the **`try catch`** expression is located) and the actual state of the environment after the **`throw`** (Δ_{catch}, that will be used on the **`catch`**). The use of each one of these elements will be introduced on the context of an appropriate rule. Finally, the format of this structure is: $\Delta_{try-catch} \rightsquigarrow N : \Delta_{catch}$

(throw)

$$\frac{\begin{array}{c}\Delta \vdash E : T \mapsto \Delta' \quad \text{stopped}(T) \quad T = N\#P \\ \langle \Omega; \Theta \rangle \in \Delta' \quad (\Delta_{try-catch} \rightsquigarrow N : \Delta_{catch-N}) \in \Theta \\ \Delta' = \Delta'_{try-catch} \uplus \Delta'_{unreachable} \quad \text{stopped}(\Delta'_{unreachable}) \quad \Delta_{catch-N} \lhd \Delta'_{try-catch}\end{array}}{\Delta \vdash \textbf{throw}\ E : \textbf{void} \mapsto \emptyset}$$

throw A `throw` expression is somewhat similar to a `return` expression, since both cause a break in the normal flow of the code even though their consequences are quite different. Remember that we require all exceptions to be declared or caught and therefore this means on this rule we must have the environment's contextual information already with the needed exception handlers registered in it.

First we simply check the correctness of the given expression which result will be the raised value at run-time. As explained above, we force this type to be in a stopped state and, since the catch is done by name, the notation must be expanded to extract the name of the throw type. Just like in the return expression, our rules follow a flow point of view and for that the carried result is `void` not the type T. For the same reason this construction leaves an empty environment since the normal flow is broken.

It is important to note that all the next conditions use the environment with the side-effects produced by the expression inside the `throw`.

The second line simply states the appropriate exception handler structures are contained inside the environment context.

For an exception to be successfully thrown we must first consider all the variables which reference will be lost as a result of the throw-catch jump. This happens since part of the environment available at the throw point (Δ') will drop all the sections that are not in the context of the respective `try catch`. Thus, we split this environment in the two disjoint environments: the one that will be lost ($\Delta'_{unreachable}$) - and all those variables contained in it must be peacefully killable so that no required behavior is left; and the environment with the same depth and environment variables as the one in the current `try catch` to which the flow will jump to ($\Delta'_{try-catch}$). This last environment will be usable in the `catch` branch and therefore all of its content may continue normally. However, a `try catch` is allowed to have multiple throws that will share the same common `catch` branch. In this case we use the same operation (reversed intersection) as in the `return` expression and therefore all these `throw` environments must be compatible with the `catch` environment of the type ($\Delta_{catch-N}$).

In spite of always mentioning a `try catch` expression as the target `catch` branch, this might not always be the case. This `catch` branch may also be completely outside the scope of the method when the type is declared in the throws list of the method. In this case the situation is much simpler since all the local environment variables will fall out of scope. However, the previous rule is still valid since the (soon to be described) class consistency check uses a similar mechanism to obtain the class fields state on each possible behavioral path. Lastly, given this `throw` is linked to the current class under verification, it will not be causing any behavioral changes to the self type as this will happen in the usual flow of the consistency path check (it just needs to save the environment state to be able to start from that point when it reaches the possible exception flow fork).

(call-throw)

$$\frac{\begin{array}{c} T \in \Delta \quad T = N[;;Methods] \quad m(...)[N_0, ..., N_n]T_r \in Methods \\ (x : T \times N\#P) \in \Delta \quad \Delta \vdash x.m(E_{arg}, ...) : T_r \mapsto \Delta' \quad \Delta' = \Delta'' \uplus (x : T \times N\#O) \\ \langle \Omega; \Theta \rangle \in \Delta' \quad (\Delta_{try-catch_i} \rightsquigarrow N_i : \Delta_{catch-N_i}) \in \Theta \\ \text{exception}(m, N_i, P) = C_i \quad \Delta_{throw_i} = \Delta'' \uplus (x : T \times N\#C_i) \\ \Delta_{throw_i} = \Delta'_{try-catch_i} \uplus \Delta'_{unreachable_i} \quad \text{stopped}(\Delta'_{unreachable_i}) \quad \Delta_{catch-N_i} \lhd \Delta'_{try-catch_i} \end{array}}{\Delta \vdash x.m(E_{arg}, ...) : T_r \mapsto \Delta'}$$

call-throw Although an exception can only be raised with the **`throw`** expression at run-time, there are other possible raise points in the type system. Given that only the method's signature is looked at in a call, all the exceptions it may raise must also be accounted for at that time.

To avoid too much duplication we take a more modular approach by making this rule use as much as possible from the previously defined call rules. For this reason, we will only describe the exception related part as the method call conditions were already presented in detail on a previous section. It is only important to know that all exceptions can only happen after the method has been called (thus all exceptions must use the resulting environment Δ' as the throw environment) and that the exception list is in the methods' signature.

Although it looks complex, this rule borrows much from the previous **`throw`** rule. This is because the handling of all those exceptions in the throws-list is basically the same as for the single throw type, it only has to be done for each single exception type separately. This is expressed with the use of index notations (i to iterate over $\{N_0, ..., N_n\}$).

The most important aspect of this rule is the behavioral side-effect suffered by the variable on the throw flow. As said above, behavioral exceptions will cause a change in the protocol of the type that causes the exception raise. In this case, it means the variable x must change its behavior on the catch environment right before the check for compatibility with environment $\Delta_{catch-N_i}$. The exception protocol (C_i) is more easily obtained by using the exception operation (definition 8) as it returns the correct protocol to be used. Note that the protocol must be the one before the call (P) as the one after it (O) only refers to the normal flow (if no exception was raised). So the main contribution of this rule is to jump the protocol of the type to that special protocol to be used only in the respective catch branch.

Definition 8 (Behavior on exception) *This operation extracts the exception behavior of the protocol. Since each id always has the same list of throwable types this operation returns the first behavioral pattern linked to the type N in the given protocol P. This operation only fails if there is no defined exception in that expression.*

$$exception(id, N, P) = P_{exception} \Leftarrow id[...|N : P_{exception}|...] \in P$$

Example 10 (Behavior on exception)

$$exception(method, Exception, method[Exception : a; b; c]) = a; b; c$$

$$exception(method, Exception, method[Exception : a]; method[Exception : b]) = a$$

internal call-throw The previous rule set does not handle one kind of situation (that might not be too obvious) that is depicted in figure 2.7. In this case the recursive internal call also uses exceptions: the `catch` handle uses a class field behavior for managing the raised `integer`.

The more generic case is an internal call-throw that must account for behavioral changes in the class' fields on each of its possible throw values. For this, the contextual exception handling structure (Θ) is insufficient and therefore we extended it to include this information: ($\Delta_{before} \ll method(\rightsquigarrow N_0 : \Delta_{fields-N_0}, ..., N_n : \Delta_{fields-N_n}) \gg \Delta_{after}$), where each $\Delta_{fields-N_i}$ is the state of the class fields after an exception of type N_i is thrown.

It is then needed to slightly modify the previous rule, to create this internal version, with the main difference residing in the fact that instead of changing the calling objects behavior (which will not happen as this is an internal call) we must change the local fields state to the one contained in the method contextual information.

(internal call-throw)

$$\frac{\begin{array}{c} T \in \Delta \quad T = N[;;Methods] \quad m(...)[N_0, ..., N_n]T_r \in Methods \\ (\textbf{this} : T_{static} \times T_{dynamic}) \in \Delta' \quad \Delta \vdash \textbf{this}.m(E_{arg}, ...) : T_r \mapsto \Delta' \\ \langle \Omega; \Theta \rangle \in \Delta' \quad (\Delta_{try-catch_i} \rightsquigarrow N_i : \Delta_{catch-N_i}) \in \Theta \\ (\Delta_{before} \ll method(\rightsquigarrow N_0 : \Delta_{fields-N_0}, ..., N_n : \Delta_{fields-N_n}) \gg \Delta_{after}) \in \Omega \\ \Delta' = \Delta'_{fields} \uplus \Delta'_{stack} \quad \Delta'_i = \Delta_{fields-N_i} \uplus \Delta'_{stack} \\ \Delta'_i = \Delta'_{try-catch_i} \uplus \Delta'_{unreachable_i} \quad \text{stopped}(\Delta'_{unreachable_i}) \quad \Delta_{catch-N_i} \lhd \Delta'_{try-catch_i} \end{array}}{\Delta \vdash \textbf{this}.m(E_{arg}, ...) : T_r \mapsto \Delta'}$$

With the intention to simplify the rules and the grammar, each `try catch` can only catch a single exception. However, since it is still possible to nest several of these expressions there is no lost expressiveness.

(try-catch)

$$\frac{\begin{array}{c} \Delta_{try} = \Delta \uplus \langle \Omega; \Theta + (\Delta \rightsquigarrow N : \Delta_N) \rangle \\ \Delta_{try} \vdash E_{try} : T_{try} \mapsto \Delta'_{try} \quad \text{stopped}(T_{try}) \\ \Delta'_{try} = \Delta' \uplus \langle \Omega; \Theta + (\Delta \rightsquigarrow N : \Delta_N) \rangle \\ \Delta_{catch} = \Delta_N \uplus \langle \Omega; \Theta \rangle \uplus (n : N\#\textbf{stop} \times N\#\textbf{stop}) \\ \Delta_{catch} \vdash E_{catch} : T_{catch} \mapsto \Delta'_N \uplus \langle \Omega; \Theta \rangle \quad \text{stopped}(T_{catch}) \end{array}}{\Delta \uplus \langle \Omega; \Theta \rangle \vdash \textbf{try}\ E_{try}\ \textbf{catch}(N\ \ n)\ E_{catch} : \textbf{void} \mapsto \Delta' \sqcap \Delta'_N \uplus \langle \Omega; \Theta \rangle}$$

```
T#a+(b;c) v; //local field v
a() throws integer{
   if( ? ){
      try{
         a(); //recursive call
      }catch(integer error){
         v.c();
         return;
      }
   }else{
      //base case
      if( ? ){
         v.a();
         return;
      }else{
         v.b();
         throw 1337;
      }
   }
}
```

Figure 2.7: An example of an internal call-throw.

try-catch The `try catch` together with the `throw` expression is a statement like control flow expression, as it causes a jump in the code from where the exception is raised to the `catch` branch.

Checking the body of the `try catch` (E_{try}) requires first to register the new exception handler (the `catch` branch) for the type N in the environment context. This is simply done by appending the handler to the exception handlers of the contextual information. Note that this action implicitly hides any previously defined handler for that type. The registration is only valid inside the `try` branch (as usual) and therefore it is not present on either the `catch` or the resulting environment (at the end of the rule). As explained above, the registration needs to store the environment of the `try catch` expression so that it is possible to calculate which variables are unreachable and those who are visible on each `throw` point. These unreachable variables are those declared inside E_{try} and therefore can not be used on the `catch` branch.

For checking the `catch` branch, besides using the throw environment state(Δ_{catch} - that has the common behaviors of all possible throws of this type that might occur in the try), it is also needed to add the catch-variable (n) that is in a stopped state, just like the allowed throwable objects.

All resulting types (T_{try} and T_{catch}) must be stopped as their behavior will not be used.

The state of the environment after a `try catch` is somewhat similar to an `if else` as there is two possible program flows that must be merged. Just like in that construction, we use the environment intersection to join the shared behavior of both the normal flow (Δ', when no exception is raised) and the `catch` branch (Δ'_N).

2.3.6 Basic types

Our language includes the normal basic types: integers, doubles, strings, booleans and also XML. We will not detail their operations as they are rather standard. These types have appropriate syntax for their construction which is only presented in the appendix.

There is no need to include syntax for normal basic operations like subtractions or multiplications since its meaning is identical of having an equivalent method call for that specific operation. In other words, in this type system syntax sugar like $1 + 2$ is removed and instead it is only used the explicit method call like $1.plus(2)$. As a side effect, this should also ease the understanding of the evaluation order of the arguments (that might have some kind of behavior).

$$\Delta \vdash \text{INTEGER} : \textbf{integer}\#\textbf{stop} \mapsto \Delta \qquad \text{(integer type)}$$

$$\Delta \vdash \text{DOUBLE} : \textbf{double}\#\textbf{stop} \mapsto \Delta \qquad \text{(double type)}$$

$$\Delta \vdash \text{STRING} : \textbf{string}\#\textbf{stop} \mapsto \Delta \qquad \text{(string type)}$$

$$\Delta \vdash \text{XML} : \textbf{xml}\#\textbf{stop} \mapsto \Delta \qquad \text{(xml type)}$$

$$\begin{array}{c} \Delta \vdash \textbf{false} : \textbf{boolean}\#\textbf{stop} \mapsto \Delta \\ \Delta \vdash \textbf{true} : \textbf{boolean}\#\textbf{stop} \mapsto \Delta \end{array} \qquad \text{(boolean type)}$$

$$\frac{N \in \Delta}{\Delta \vdash \textbf{null} : N\#\textbf{stop} \mapsto \Delta} \qquad \text{(null type)}$$

All basic types have a **stop** use pattern so their use never changes the checking environment. This empty behavior also allows any method to be freely used (as expected there is no restriction for calling methods of any basic type).

The **void** type expresses an empty type. Therefore, even though it is not syntactically blocked, there is really no use in the language for a variable/argument with it since it can not legally receive any content (and thus will always stay **void**).

2.3.7 Class type

These class types are responsible for storing all relevant class related information. Therefore, this rule is only intended to build these structures. Furthermore, there are some additional conditions which are not directly shown, namely: all method's return type are forcefully owned (T°) and any method's parameter has its ownership value set when the **owned** keyword is present in that argument (otherwise its type is non-owned $T^{\bullet}$).

(class declaration)

$$\frac{\begin{array}{c} Name \notin dom(\Delta) \\ Fields = \{(x_0 : T_0), ...\} \\ Methods = \{(M_0(T_{arg_0}\ \ n_0, ...)[N_0, ...]T_{ret_0}\{E_{body_0}\}), ...\} \\ \forall M_i \in Method : M_i = Name \Rightarrow M_i(...)[\emptyset]\textbf{void} \\ \forall m : m[N_0, ..., N_n] \in Methods \land m[N_0, ..., N_m] \in Pattern \Rightarrow m \geq n \end{array}}{\begin{array}{l} \Delta \vdash \ \textbf{class}\ \ Name\{ \\ \ \textbf{usage}\ \ Pattern \\ T_0\ \ x_0; \\ ... \\ T_{ret_0}\ \ M_0(T_{arg_0}\ \ n_0, ...)\ \ \textbf{throws}\ \ N_0, ...\ \ \{E_{body_0}\} \\ ... \\ \} :\ \textbf{void} \mapsto \Delta \uplus Name[Pattern, Fields, Methods] \end{array}}$$

This construction is intended to build a class type while obeying some additional rules in order to make sure the given usage protocol is consistent with the class' methods, namely:

- constructors are normal methods with a **void** return type, the same name as the class and no throwable exceptions.

- each field name must be unique inside the $Fields$ set.

- each pair (method name, number of arguments) must be unique inside the $Method$ set.

- each transition of the usage protocol (id's) must refer to at least one existing method with the same name (which excludes constructors).

- every time an id appears inside the protocol it has to declare the same number of possible throwable exceptions and each method with the same name as id must only throw a subset of that set of exceptions (never more).

- as implied by the grammar, each class can only have one usage protocol that is assigned to the type after a new instance is create.

2.3.8 Class Consistency Check

As said above, the essential verification happens on a class level. This means that any expression type checking only occurs inside the context of a class verification. However, it is not possible to test each method independently as they may include behavioral uses of the class' fields that must be considered. These uses are impossible to verify without a concrete usage scenario that is given by the class' protocol. As such, this class consistency check is the one responsible to assure the correction of a class as seen from a higher level and also to do the transition to lower ones. Consequently, in this section we will present the rules that travel through the class' behavior and test if each allowed method's body in that transition is consistent with the larger class level (i.e. with the complete protocol).

As a result, these rules cover each possible protocol construction until they reach a single labeled transition. At that point, they must go inside the method to assure all its fields behavioral uses are correct in that given protocol context.

The validation of any class field with a non stop behavior poses an interesting problem since its use may be split among any of the class' methods. As a non behavioral method is free to be called in any context, we decided to prohibit it from changing fields (either by calling some of the object's behavioral methods or just by making a normal assign, etc). Thus, only behavioral methods are allowed to modify these variables since their context is given by the class' usage protocol. Therefore, it is needed to do this class consistency check to verify if the fields' use is correctly split among the behavioral methods in a coherent way. Thus, to guarantee the correct use of these local class variables, it is needed to consider not only the code of the method but also it is execution context and so verify the consistency of the method's body according to all valid paths that the usage protocol might allow for it to be called from.

$$\text{(class consistency check)}$$
$$\frac{\Delta_{fields} = \{(f : T^{\circ} \times \text{stop}(T)) : f \in Fields\} \qquad \Delta \uplus \Delta_{fields} \vdash Protocol : \text{OK} \mapsto \Delta \uplus \Delta'_{fields} \quad \text{stopped}(\Delta'_{fields})}{\Delta \vdash Class[Protocol; Fields;] : \text{OK} \mapsto \Delta}$$

This kind of check differs from previous ones as we try to reach an "OK" state which confirms the correction of the fields use. We first start by placing all fields in a stopped initial state and append it to the checking environment (that contains other previously defined types). This is similar to a variable declaration but for each single one of those field variables. The rule then proceeds to travel through all possible protocol paths. Note however, that since any constructor will modify the initial state of these fields its check must be done before starting this procedure. At each end we must then check for a correct field behavioral termination as all those objects may be destroyed at that point.

$$\text{(protocol sequential path)}$$
$$\frac{\Delta \vdash P : \text{OK} \mapsto \Delta' \quad \Delta' \vdash Q : \text{OK} \mapsto \Delta''}{\Delta \vdash P; Q : \text{OK} \mapsto \Delta''}$$

$$\text{(protocol choice path)}$$
$$\frac{\Delta \vdash P : \text{OK} \mapsto \Delta' \quad \Delta \vdash Q : \text{OK} \mapsto \Delta'}{\Delta \vdash P + Q : \text{OK} \mapsto \Delta'}$$

$$\text{(protocol repetition)}$$
$$\frac{\Delta \vdash P : \text{OK} \mapsto \Delta}{\Delta \vdash P* : \text{OK} \mapsto \Delta}$$

$$\text{(protocol recursion)}$$
$$\frac{\Delta \vdash T\{b/\&b(T)\} : \text{OK} \mapsto \Delta'}{\Delta \vdash \&b(T) : \text{OK} \mapsto \Delta'}$$

Path traveling follows a simple logic: the sequence construction must be correct on the left side and will be followed on the right with the left state as initial; both paths in a choice must be correct; a repetition is only correct if it remains in the same state; a recursion (in the protocol) is obeyed if the unfolded protocol is correct.

(behavioral method check)

$$\frac{\begin{array}{c} method = method(T_0\ \ n_0, ..., T_n\ \ n_n)T_r\{E\} \\ \Delta_{args} = \{(n_0 : T_0 \times T_0), ..., (n_n : T_n \times T_n)\} \\ \Delta_{before} = \Delta \uplus \Delta_{fields} \uplus \Delta_{args} \uplus (\textbf{this} : Class\#\textbf{stop}^{\bullet} \times Class\#\textbf{stop}) \\ (\Delta_{before} \ll method(\rightsquigarrow N_n : \Delta_{fields-N_n}, ..., N_n : \Delta_{fields-N_n}) \gg \Delta_{after}) \in \Omega \\ \Delta_{before} \uplus \langle \Omega; \Theta \rangle \vdash E : T \mapsto \Delta_{after} \uplus \langle \Omega; \Theta \rangle \quad \text{stopped}(T) \\ \Delta_{after} = \Delta \uplus \Delta'_{fields} \uplus \Delta'_{args} \uplus (\textbf{this} : Class\#\textbf{stop}^{\bullet} \times Class\#\textbf{stop}) \quad \text{stopped}(\Delta'_{args}) \\ (\emptyset \rightsquigarrow N_i : \Delta_{fields-N_i}) \in \Theta \\ \Delta \uplus \Delta_{fields-N_i} \vdash C_i : \text{OK} \mapsto \Delta \uplus \Delta'_{fields-N_i} \quad \text{stopped}(\Delta'_{fields-N_i}) \end{array}}{\Delta \uplus \Delta_{fields} \vdash method[N_0 : C_0 | ... | N_n : C_n] : \text{OK} \mapsto \Delta \uplus \Delta'_{fields}}$$

This (rather complex) rule is the one from which the previous expression, call, etc. typing rules are called and therefore it must set all the needed context structure and arguments before it can check the body. First, all arguments must be packed with a ready to use state. This is done by just making the dynamic state the same as the static one (just like if it had received a valid content after an assign). It must also add to the environment the this pointer which is in a stopped state so that it cannot be returned or changed.

All contextual information (methods context, and declared exceptions) are then added to the environment before the body of the method is checked. Afterwards all arguments must be in a stopped state to assure their use was completed. Lastly it only remains the need to check eventual exception paths for each type in the exception list, before continuing with the protocol's normal path.

(non-behavioral method check)

$$\frac{\begin{array}{c} \Delta_{fields} = \{(f : T^{\bullet} \times \text{stop}(T)) : f \in Fields\} \\ \forall m \in \{m : m \in Methods \wedge m \notin Protocol\} \Rightarrow \Delta \uplus \Delta_{fields} \vdash m : \text{OK} \mapsto \Delta \uplus \Delta_{fields} \end{array}}{\Delta \vdash Class[Protocol; Fields; Methods] : \text{OK} \mapsto \Delta}$$

The methods which do not belong to the protocol must also be checked. These methods can be freely called from any context, and therefore, to avoid possible interferences with the behavioral methods, all variables have a stopped protocol and read only permissions. The main differences with the previous rules is that this one requires the fields to be initiated directly into a stopped state and each method has to be checked independently (not inside a context path). Exceptions are handled as if the fields state remained stopped (these exceptions cannot cause behavioral changes).

2.3.9 Containers

Containers are a special kind of class that can hold an arbitrarily large group of objects. Thus, these objects are kept by the container until they are either removed or the container is destroyed. Since there is no handler for the destruction of any remaining object inside the container it is necessary for the holding type to take into account this possibility. Therefore, all inserted objects must be in a stoppable state so that their behavior is completed even if they are never removed.

As with all object-holding constructions, the holding type must be a partial type of some known type to guarantee there is at least some protocol that can obey the declared behavior.

$$\text{(container declaration)}$$

$$\frac{N\#P \in \Delta \quad x \notin dom(\Delta) \quad N\#Q <: N\#\textbf{stop} \quad N\#Q \prec: N\#P}{\Delta \vdash \text{Container}\,\langle N\#Q\rangle \quad x : \textbf{void} \mapsto \Delta \uplus (x : \text{Container}\,\langle N\#Q\rangle)}$$

This declaration has many similarities to the variable declaration with the main difference residing in the content's allowed behavior. The second condition (with the partial type) is for the same kind of restriction as in variables, however the requirement to be a sub-type of **`stop`** is to model the necessity for each holding element to be in a stoppable state.

$$\text{(container read element)}$$

$$\frac{(x : \text{Container}\,\langle T\rangle) \in \Delta}{\Delta \vdash x.readElement() : \text{stop}(T) \mapsto \Delta}$$

$$\text{(container remove element)}$$

$$\frac{\begin{array}{c}(x : \text{Container}\,\langle T\rangle) \in \Delta \quad \langle\Omega;\Theta\rangle\ \in \Delta \\ \Delta = \Delta'_{try-catch} \uplus \Delta'_{unreacheable} \quad \text{stopped}(\Delta'_{unreacheable}) \\ (\Delta_{try-catch} \rightsquigarrow Empty : \Delta_{Empty}) \in \Theta \quad \Delta_{Empty} \lhd \Delta'_{try-catch}\end{array}}{\Delta \vdash x.removeElement() : T^{\circ} \mapsto \Delta}$$

$$\text{(container add element)}$$

$$\frac{\Delta \vdash E : T^{\circ}_{new} \mapsto \Delta' \quad (x : \text{Container}\,\langle T\rangle) \in \Delta' \quad T_{new} <: T}{\Delta \vdash x.addElement(E) : \textbf{void} \mapsto \Delta'}$$

Since with the read operation the element still remains inside the container, and so to avoid any possible future interferences, this method returns a stopped view of the stored object. On the other hand, the remove operation also guarantees no one else has the ownership of the type and thus it is legal to return an owned type with the container's holding protocol. Trying to remove an element from an empty container will cause an exception to be thrown. Returning null on empty is not a valid alternative since this type is implied to have a stopped protocol which might not be the same as the holding type behavior.

Also note that there is no explicit constructor for this class and that its use pattern is always implied **`stop`** after the declaration. Because the use of containers is limited to class fields in the prototype, they will not be developed much further in this formalization.

2.4 Remarks

Classes are the essential structure of our type system. As such, our behavioral reasoning is only applied to them as they form the central block that assures the correctness of the whole system. These classes are allowed to contain (always private) field variables as well as (always public) methods. Besides this information, a behavioral type also needs to have a usage protocol. Therefore, each class' correct behavior is described in a special regular expression like protocol that limits the availability of its methods. Consequently, it is insufficient to check each of its methods independently as the behavior of a fields is split among them in a special context. This context is directly related to the expected behavior, which leads to our consistency checking rules that generate the correct behavioral context in which to verify the expression contained inside a method.

In this chapter we have presented a behavioral type system whose central concept is to combine a normal type (Java-like object type) with a finite deterministic automaton described by an regular-expression-like protocol.

This protocol limits the use of some methods (behavioral methods) to a valid context given by the class' usage protocol (that describes the allowed sequence of calls the class must obey). All methods that are not in that condition are thus considerate normal or non-behavioral and thus can be freely called without any restriction.

In order to ensure the behavior is obeyed and completed, each class instance (object) must be stored in variables that control its use by monitoring the flow of calls. This dynamic protocol of the next valid behavior changes on every legal transition (method call, etc). Note the definition of behavioral methods is linked with the usage protocol and not the dynamic protocol, therefore the set of behavioral methods never changes after the class' definition.

To avoid possible interference in the use of aliased objects, only a single variable can keep the behavior of a given object. This means there is a kind of behavioral linearity in the sense only a single variable can know the current allowed behavior of any object, however it is possible to have more than one variable holding a reference to that same object as long as its allowed behavior is seen as empty (**`stop`**) and thus can never cause interferences with the real owner. Incidentally, this kind of ownership is required in specific contexts (returning a value, etc) and is represented by the **`owned`** keyword.

Class fields pose a specific problem in order to avoid possible illegal access combinations, namely the use of these variables is usually split among different methods. However, the class' protocol gives the expected context for each behavioral method allowing for a consistency check to ensure their use is correct with their state in all possible sequences of calls allowed by that protocol.

Finally, this formalization of the type system is intentionally simplified to ease comprehension and the writing of the rules. However, the implementation in the prototype has expanded some of these rules into a more user friendly version by grouping common constructions or including more practical deviations of the rule's intention, some syntax sugar and even some additional features like static variables (which are always in a stopped state - and thus without any behavior left - since their use can appear freely anywhere).

Also note that this type system is only intended to account for untrapped errors even though trapped ones (zero divisions, null pointer de-referencing, out of memory errors, etc.) can lead to an abnormal program termination with incomplete behaviors. There are, however, other type systems that can be used to avoid some of these situations.

In the following chapters we will introduce the prototype with examples and lengthy descriptions of the algorithms that will implement the rules previously presented.

3

Examples

This chapter contains three examples that show more practical uses of the developed language and type system. Although some of the extensions to the typing rules will only be introduced in a following chapter, these examples should give a more clear notion of the general functionality that we intend to reach. This also means the grammar used in here is much more flexible than the one presented before even though the general felling of it should remain similar (the full grammar will only be listed in appendix A).

In the first example (section 3.1), we model the use of a (fake) file. This is a very simple example as it follows the normally expected method structure similar to any other file classes while also providing the additional notations needed for the type system to check the correct flow of calls. Therefore, with this type system it is then possible to assure all opened files are closed before the respective file object is destroyed.

The machines example (section 3.2) is intended to simulate the environment in a simple factory. In it there are several machines with different capabilities which will modify a give work block in particular ways. Although there are still some important expressiveness limitations that are left as possible future work, we believe our type system allows for some behavioral modularity that in turn makes the use of these (virtual) machines safer (not only in the normal typing way but also by making sure each machines is not driven into a possible erroneous state).

The final example (section 3.3) is probably the most interesting one from a behavioral point of view. It makes use of partial types in order to save incomplete purchases of several different types of orders. Therefore, these all share a common suffix by allowing a review and a buy operation. Each one of those orders also has specific requirements in terms of normal flow (for instance, renting a vehicle might require the purchase of some kind of insure). Although all these services are local, our prototype would also allow for their distribution into different servers while only needing their addresses (URL) to work in the same way as presented in here.

Note that some of these examples use a simple additional construction of the language that allows for the creation of labels for a specific section of a class' usage. These appear after the keyword "use" and are accessible by using the "$" sign followed by the previously declared label. They are meant to avoid the repetition of commonly used sub-behavior by assigning it a small user-defined name. As in previous examples, "?" is used to express an irrelevant non-behavioral expression that will not disturb the normal flow of the example and therefore is hidden to simplify the code.

3.1 Files

This is the short presentation of the file example, the full listing is on appendix D.1.

We decided to simplify the protocol while also respecting the most common uses of files. This means that some additional possibilities (like for example to allow a file - opened with read and write permissions - to still be writable after a read call has failed) are not present as they would make the usage protocol too complex and hard to understand (since it needs to attach different exceptions on each read call). Future work could be done to improve the readability of these protocols with constructions that can be split in different and smaller protocols which would then help in this situation.

Our file protocol restricts the allowed method calls to be consistent with the permissions given by the chosen open method. This means a read only file will not be allowed to call any of the write methods. Also, even if an error occurs in the use of a file, it still must be correctly closed as expressed by the `IOException` exceptional behavior. We also allow for a non-existing-file error to be recoverable by changing the target path and retry from the beginning of the protocol.

On the `Main` classes we show several possible correct uses of the `File` usage protocol.

```
interface File{
    usage &start((
       ( openRead ; read* ) +
       ( openWrite; write* ) +
       ( openReadWrite; (read+write)* )
       ; close
       )[ openRead, openWrite, openReadWrite
                -> FileNotFound: stop+(changeFile;start) |
          read, write
                -> IOException: close ] )
    /* ... */
}

class Main{

   readSomeMore(File#(read*)[IOException:stop] file) throws IOException {
      file.read();
   }
   loopOpenRead(
      File#&start( openRead [ openRead
            -> FileNotFound: changeFile;start ] ) file){
      repeat{
         try{
            file.openRead();
            return null;
         }catch(FileNotFound exception){
            file.changeFile(file.name()+"0");
         };
      }
   }

   main(){
      File file = new FakeFile();

      if( Lib.random() >= 0.5 ){
         loopOpenRead(file);
      }
      else{
         try{
            file.openRead();
         }catch(FileNotFound exception){
            return null;
         };
      };

      try{
         while( false ){
            file.read();
         };
         readSomeMore(file);
      }catch(IOException exception){ };

      file.close();
   }

}
```

The following class shows situations in which the typechecker detects behavioral inconsistencies.

```
class Fails{

   error1(){
      File file = new FakeFile();
      file.openRead(); //ERROR: unhandled exception ’FileNotFound’

      try{
         file.read();
         file.read();
      }catch(IOException e){}
   } //ERROR: incomplete behavior in 'file'

   error2(File#&start( openRead [ openRead
            -> FileNotFound: changeFile;start ] ) file){
      repeat{
         try{
            file.openRead();
         }catch(FileNotFound exception){
            file.close(); //ERROR: illegal call to 'close'
         };
      } //ERROR: invalid 'file' behavior for loop
   }
}
```

3.2 Machines

In this example (full listing on appendix D.2) a generic block is modified by a set of machines in a particular sequence given by a specific blueprint class. Thus, in order to correctly build some object it defines a sequence of operations that must be applied to one or more blocks. Unfortunately, since no dynamic query-by-protocol is available in the language (see future work, section 5.1) every machine use is frozen in the code as each one of them has particular and well defined features. The flow of the object through the factory production line also becomes somewhat “ugly” as we also do not have parametric types that could improve the code's modularity (see section 5.1) and instead must rely on using these machines in argument form to simplify the code.

Nonetheless, this example shows some interesting features of the language as it controls not only the use of these machines but also some of the block's abilities in order to avoid possibly erroneous operations (like welding a block after it has been painted, etc).

```
class Block{
   usage (cut+bend+weld+paint)*

   /* ... */
}

class Bender{
   usage ((lock;(bend*+twist);release)*;standby)*

   /* ... */
}

class Painter{
   usage (( enter; ((rotateLeftGun+rotateRightGun)*;paint)*; exit)*; standby)*

   /* ... */
```

```
}

class CarBlueprint{

   bend(Bender#lock;(bend*+twist);release bender, Block#bend* b,
      integer angle, integer amount){

      bender.lock(b);
      if( angle > 180 ){
         bender.twist(b,angle,amount);
      }else{
         while(angle > bender.maxBend() ){
            bender.bend(b,bender.maxBend(),amount);
            angle -= bender.maxBend();
         };
         bender.bend(b,angle,amount);
      };
      bender.release();
   }

   object build(BlockWharehouse w, Factory m){
      Bender bender;
      Painter painter;

      try{
         bender = m.getBender();
         painter = m.getPainter();
         }catch(EmptyMap error){
         //intentionally stops every machine found
         return null;
      };

      Block wheels = w.getBlock();
      Block car = w.getBlock();

      /* ... */

      bend(bender,wheel_fr,90,100);
      bend(bender,wheel_fl,90,100);
      bend(bender,wheel_br,90,100);
      bend(bender,wheel_bl,90,100);

      /* ... */

      //paint here...
      painter.enter(car);
      painter.rotateLeftGun(90);
      painter.rotateRightGun(90);
      painter.paint("black",3);
      /* ... */

      Block#stop done = painter.exit();

      m.returnBender(bender);
      m.returnPainter(painter);

      return done;
   }
}
```

3.3 Purchase

This example has many different types of orders, in here we will only show two (the full listing is on appendix D.3). All orders' behavior end in the same way as they all allow for the review and purchase of the product. This is deliberately made so that they all can be stored in the same container based on this partial type. Any order can be kept in an waiting state inside the container for an undefined amount of time. These orders will be implicitly canceled if the program ends as they model a kind of wishlist.

Each different type of order requires a specific sequence of calls so that all their requirements are filled. For instance, on travels it requires to chose from either a specific travel package or select a flight and hotel. These types can also be combined or nested together as in the case of the `TravelOrder` requiring a `FlightOrder` when booking a flight.

```
interface Order{
   usage review*;buy?

   string review();
   buy();
}

class TravelOrder{
   usage (packageAlaska+packageArtic)[SoldOut: stop]+
         (flight;hotel); (review*; buy?)

   use done = review*;buy?

   flight(owned FlightOrder$done order) { }

   /* ... */
}

class FlightOrder{
   usage userData[NoFlyList: stop];
      &start(
         ( flightNumber[InvalidFlight:start+stop] ;
           &seat(flightSeat[InvalidSeat:seat+start+stop]) )
            +
         &choose( (destination;origin)[InvalidPlace:choose+stop] );
         returnFlight? ;
         insurance? ;
         (review*; buy?)
      )

   use done = review*;buy?

   /* ... */
}
```

Some fail cases:

```
class Fails{
   error1() throws NoFlyList{
      TravelOrder travel = new TravelOrder();
      FlightOrder order = new FlightOrder();

      order.userData(...);
      //ERROR: on 'NoFlyList' throw -> 'travel' still has incomplete behavior

      //...
   }

   error2() {
      TravelOrder travel = new TravelOrder();
      FlightOrder order = new FlightOrder();

      try{
         if( ? ){
            travel.packageAlaska(...);
            order.userData(...);
         }
         else{
            order.userData(...);
            travel.flight(...);
            travel.hotel(...);
         };

         travel.buy(...);
      }
      catch(SoldOut error){
         return null; //ERROR: 'order' still has incomplete behavior
      }
      catch(NoFlyList error){
         //ERROR: 'travel' with incompatible catch behavior
      }

      //...
   }
}
```

4

The Prototype

The type system presented in the previous chapter was implemented in a prototype. As with any proof-of-concept implementation, the main objective is to make the code clear and simple even if this means choosing less efficient algorithms. Therefore, some optimizations are left undone and minor tricks that could improve the overall complexity of some of these rules are not used. Hopefully this should make the resulting implementation easier to read and understand.

Also, as a result of this code being the continuation of a previous work [26] it carries some decisions and requirements that might look out of place for the specific set of objectives we set for this dissertation. Namely, there are some additional syntax (related to concurrency) which will not be detailed and the whole apparatus of creating a parser, interpreter and type system could have been cut to just the last if this were a completely fresh project.

As mentioned before, this is not the exact same version of the grammar and rules presented in a previous chapter as those would be rather restrictive and limited to be used directly. Instead, we have extended that basic set (while retaining the general intention of that system) by adding some syntax sugar and combining typing rules together. This full grammar will not be presented in this text but it is listed at the end (see appendix A).

Conclusively, the following sections contain the description of the most important algorithms of the prototype. Pseudo-code is used for better readability while also providing references to the appropriate part of the source code (file and line) that has the real implementation. We start by briefly introducing the central technologies used throughout this project (all freely available) that might be useful to anyone interested in understanding or changing the code. After that, we present a small list of the main extensions and differences to the formal type system. We then proceed to introduce the type checker describing all of the created logic before shortly explaining the interpreter and the distribution mechanism.

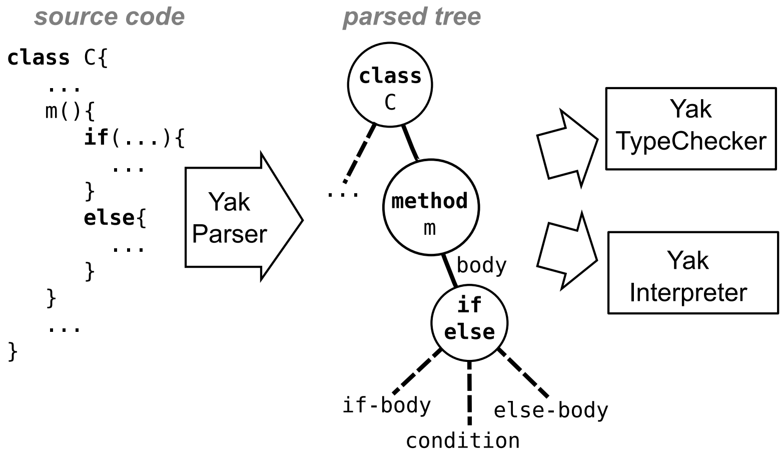


Figure 4.1: General program flow.

As explained in figure 4.1, the general program follows a very simple and modular approach. The parser builds an AST tree with the program structure (see appendix C.2 for the run options to draw it), this tree is then passed on to the typechecker that analyses the program's correctness or its evaluation is calculated if it is handed to the interpreter. The visitor design pattern is used to visit each of those (relevant) nodes. Complexity wise, we will only comment on the computation time of each visit method while leaving the overall complexity analysis to the remarks section.

4.1 Technologies

This prototype was developed using a group of freely available technologies, namely:

Java [30] programming language, developed by Sun Microsystems. It is a widely used language, with a large and well documented set of libraries and support for OS independent code compilation that runs on any java virtual machine regardless of the underlying hardware architecture. Another relevant aspect is the possibility to easily extend and add features to web servers using Java Servlets.

Jetty [31] It is a free and fast implementation of the Java Servlet API, with special attention to resource efficiency and a general small run-time footprint.

JavaCC [32] This java compiler compiler allows to quickly create parsers and also has a small set of tools for the Eclipse IDE [33] (like a plug-in [34]) to help the development of new grammars.

JDom [35] Although the Java programming language already includes support for XML manipulation in its class libraries, this library simplifies much of work needed and also enjoyed from good experiences from its use in previous projects.

JUnit [36] testing framework was used for all unit testing needs in order to ensure any new feature or bug fix did not cause any regression or the introduction of some new abnormal behavior.

dk.brics.automaton [37] This efficient and fast automaton provides a large set of operations making it the ideal base for adding the extra functions required to the type system.

4.2 Main changes to the formalization

As said before, this typechecker expands the previously presented type system rules into a more user friendly version while still maintaining the core idea of each one of them. In this section we will present the most important changes and how they were addressed in the prototype.

concurrency The complete grammar has some additional concurrency construction (fork of the program flow and parallel composition of statements) that do not have an appropriate typing rule. Therefore, and to avoid any possible interferences all behavior and exception throwing in these expressions is strictly forbidden.

discard Instead of forcing all values to be stored in variables, we control all newly created references so that, at the end of a statement their behavior is either completed (stopped) or was saved somewhere (in a variable, returned or passed as an owned argument).

non-owned These variables are no longer constants as we control their flow in the program code. Thus, some rules where extended to account for changes in the owned flag on different branches, etc.

nulls Instead of just relying on the subtype system, we allow for all stack variables to receive the null value as a stopped type. This is also the variable's initial content. Note that this only applies to stack variables, not arguments or returned values.

fields Classes with stopped usage protocols can have their fields content changed in any method as long as that variable's static type has `stop` behavior. This is only valid because there is never the chance to occur an inconsistency derived from a behavioral method (as, since it is a stopped usage protocol, there are none). Thus, in this case, the use of a behavioral field ends up being as if it were just a normal stack variable without behavior.

interfaces As a simple abstraction mechanisms, interfaces can be used as abstract classes. Therefore, interfaces are simply classes without any constructor and whose method's have no actual body.

static variables We include simple static variables that always have a stopped behavior as their use can appear in any context.

4.3 Implementation of the Typechecker

The developed typechecker is intended to validate any program written in the **yak** language in terms of its behavioral properties. Therefore, it does not try to solve inconsistencies that may arise from other types of errors (which still may occur at run-time), namely: null pointer exceptions, broken remote connections (when using remote objects) or even invalid generated XML code (when using invalid names in the XML constructor which will throw a JDom exception). Also note that the rules defined do not cover the whole (implemented) grammar as they do not handle the concurrency constructions. Therefore the two existing expressions, "**fork** E" and "E|E", have all behavioral calls and exceptions blocked from use in order to avoid possible interferences with the rest of the typing rules.

The typechecker was implemented using the visitor pattern to transverse the useful AST nodes of the parsed tree. For that reason, the significant constructions are usually linked to a specific visit method (in the typechecker) for the corresponding node type.

Before describing each of the typing algorithms we will first introduce some of the used structures and operations. Thus, we start by a detailing the automaton and its simulation operation (including some examples), followed by a briefly description of the environment snapshot method. We then proceed to show each one of the type checking code. The pseudocode used is intended to serve as an introduction to the real implementation and therefore it closely resembles the final Java code while hiding some of the more complex details.

Only the most important aspects of the prototype are presented. We refrain from explaining some (mostly) simple code that does not add much new information to what was already described in the formalization chapter or code that does not include any new relevant idea. For example, the subtyping algorithm is very similar to the one presented in the article "Subtyping Recursive Types" [38] and as such the only really new restrictions were described in the formalization chapter. Consequently, this algorithm is not described in here.

4.3.1 Basic mechanisms

The automaton

All declared protocols are internally translated into finite deterministic automata. The used automata are basically a slightly modified version of the dk.brics.automaton [37]. Some minor tricks were needed in order to use labeled transitions and exceptions and also some other changes were necessary as a consequence of the specific context of this prototype (like using sections of the same automaton within an intersection operation, etc). We will only present the newly implemented functions. However, this library provides a large set of well known operations that were used but will not be shown in here since their detailed description can be found elsewhere [4].

An issue on how Java handles servlet libraries lead to the need of all the implemented extensions to be placed inside the same package as the rest of the automaton library. Therefore, operations that were modified from the original automaton source code (in this case mostly just the automaton intersection) were placed in `dk.brics.automaton.ModifiedOperations` while newly added operations are in `dk.brics.automaton.YakOperations`. Finally, it was also necessary to make a special construction method to delay the insertion of recursion and exceptions links and produce the intended automaton. There is also a "quick and dirty" automaton viewer that, although far from pretty, is sufficiently functional to serve as a debugger tool for the creation and simulation algorithms (details on how to use it are in appendix C.2).

The YakAutomaton class is mostly an extended version of the normal automaton that stores its history (called transitions), the initial creation expression and calls the correct version of the operations. The saving of the transition history is only needed because we do not have an appropriate algorithm to translate the currently allowed automaton into a readable protocol. We also chose to make a small cache of all newly created automata and called intersection operations as our implementation rules may cause some possible repetitions (see recursion) that will benefit from reusing these structures.

Note that although the degree of freedom allowed by the protocol syntax may cause the creation of empty automata (without a reachable accept state, example $\&x(r;x)$) this situation is flagged as an error since it will always be impossible for any program to satisfy this behavior.

The complete listing of all automaton protocols is in appendix B and as such, in the remaining of this section we will only briefly detail the intricacies of the most important automaton constructions: exceptions and recursion.

Our automaton construction algorithm can not use directly the same construction code as in the automaton library since we don not follow a feasible creation flow compatible with their construction methods. For that reason, we first create a non deterministic finite automaton that is then translated to their structures and determinzed/minimized. All of our side of the construction is not particularly optimized as we focused more in the typechecker rules and as such there is room for improvements in this section of the code.

The following constructions can be seen as separate sub-automata that are then attached to the larger one when all the branches have completed their construction process.

Exceptions Exceptions are just normal transitions but whose label is only satisfied in the context of a **`throw`** statement. The special behavior is always linked as a separate independent automaton, this means that at the end of its behavior there is no link back to the normal automaton (the normal and exceptional behaviors are completely disjoint). There is also a normal exception declaration (i.e., non-behavioral) for these protocols which simply loops to the same state after a **`throw`** as the behavior remains the same.

The example in figure 4.2 depicts the automaton produced by the protocol

$$normal[Error0|Error1 : exceptional]; end$$

that has a method called $normal$ that may either throw a normal (non-behavioral) exception ($Error0$) or a behavioral one ($Error1$). After the first method is called it must proceed to call

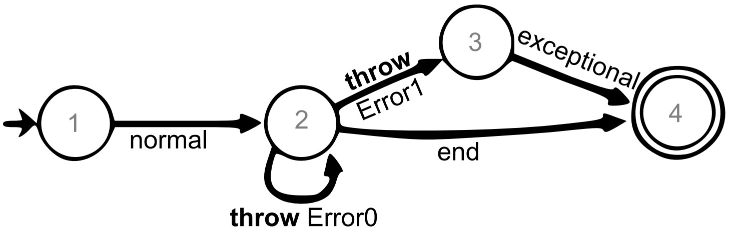


Figure 4.2: Automaton for protocol $normal[Error0|Error1 : exceptional]; end$

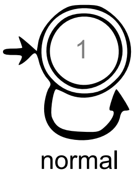


Figure 4.3: Automaton for protocol $\&label(normal; label + \textbf{stop})$

the end method except when its behavior changes after the exception $Error1$ is raised. In that case, only the $exceptional$ method should be called. Since this is just a very simple protocol, it is possible to list all valid sequences of behaviors:

- $normal; end$ - when no exceptions are thrown;
- $normal; Error0; end$ - when the non-behavioral exception $Error0$ is raised;
- $normal; Error1; exceptional$ when $Error1$ is flagged in the initial method.

Recursion The theoretical idea of protocol recursion is to allow the repetition of a specific section by unfolding it into the labeled position. However, to fully respect this it would be required a context independent grammar which has known and complex implementation problems. Therefore, the presented solution is a much more limited and simple subset of this idea by instead just linking the labeled position back to the labeled automaton node. This pointer scheme, although less flexible, should be sufficient for most cases specially since this construction's main purpose is to allow recursion in the behavior of exceptions (in other words, allow the repetition when/if an exception is raised).

This means that some recursive protocols (even if syntactically correct) may produce erroneous or unsatisfiable automata. For example protocols $\&x((a; a; a)?; x)$ and $\&x(a + stop; x)$ are not correctly constructed by our protocol recursion algorithm as the loop linking operation breaks the acceptance state.

Figure 4.3 shows a recursive protocol that is correctly translated into a finite deterministic automaton by our construction algorithm. This example is actually equivalent to the non-recursive protocol $normal*$ but there are cases in which there is no possible conversion among the notations. Incidentally, these two notations can be mixed together which is not recommend as the produced automaton is kind of hard to foresee without detailed knowledge of the implementation.

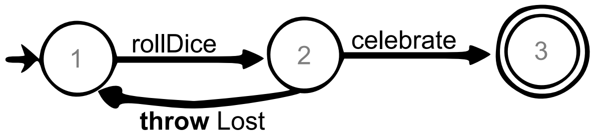


Figure 4.4: Automaton for protocol $\&start(rollDice[Lost : start]; celebrate)$

The main argument for the inclusion of a recursive construction for these protocols is to model the situation represented in figure 4.4. It described the behavior of a compulsive gambler (with infinite money) that plays dice forever until he wins. Therefore, in a win situation it then calls the $celebrate$ method but if the $Lost$ exception is thrown it goes on to re-try the game again.

The remaining construction operations are detailed in appendix B.

Even though the construction of this automaton could have some improvements, the expected code complexity would still remain exponential in the number of states as a consequence of the determinization operation. Therefore, we try to minimize the number of new constructions by caching these automata as much as possible and only creating new ones mostly in the beginning of the typecheck.

Simulation algorithm

This operation will expand the normal automaton transition in order to allow another automaton to serve as the forwarding term. As a result, this operation can also be used to test if the smaller automaton (which might be more than just a single transition) can be temporarily used as a subset of the larger one (upon which the simulation operation was called on). At the core of the simulation algorithm is the matching function that starts with an initial state from each automaton and goes on to try and match the whole reachable states. After this function has forwarded the original automaton with the smaller one, it then must intersect all of the end nodes to create an automaton that shares all those valid behaviors.

Note that our implementation requires not only the automaton to be deterministic but also minimized.

The more generic pseudo-code of the implementation is:

```
s1.simulate(s2){
   match (super = s1.startNode) with (sub = s2.startNode)
      super.nomalTransitions contains all sub.normalTransitions;
      sub.exceptionsTransitions contains all sub.exceptionsTransitions;

      for each common transition
         continue matching transition.destination nodes;

   return intersection of all resulting (end) nodes;
}
```

Name wise, we call the automaton that suffers the simulation operation ($s1$) the larger automaton or from a sub-typing point of view the super-automaton (as it is related to the supertype). The automaton that is used as the argument of it ($s2$) is in turn called the smaller automaton (as it must be contained inside the previous one) or the sub-automaton.

On each pair of possible matched states it then tests their set of transitions to determine if they are compatible. This compatibility is based on the subtyping principle where the matching automaton must be able to be safely used as a replacement of the base one.

Therefore, these two following rules expand on that idea:

normal transitions The super-automaton must at least have all the transitions of the sub-automaton. Thus, any choice in the sub-automaton has to be matched to one in the larger automaton. This is not a reflective properties as the inverse situation is not required as any transition in the larger automaton might not appear in the smaller one (this will just cause any of those extra choices to become hidden). (remember $a + b + c \xrightarrow{a+b}$ **stop**)

exception transitions Exception wise the transition situation is the opposite of the previous rule: the smaller automaton must have all the exception transitions of the super-automaton but it is also allowed to declare more than that (intuitively this means the super-automaton can not surprise the sub-automaton with unexpected exceptions but the last one can account for additional errors that the super-automaton will never throw). (remember $a[N : b] \xrightarrow{a[N:b|M:c]}$ **stop**)

The remaining of the matching function just needs to group every matched state and unfold any possible repetition. As a consequence of the specific way we build these automata, any exception transition in the beginning of the match must be ignored as it is referring to an impossible situation of a threw exception without a method call. This only happens when we reuse a subsection of an automaton without actually detaching a copy of it. Since it is just an implementation note, it will not appear in any of the examples.

All the end states of the smaller automaton are potentially end states after the simulation has ended. As such, by getting the appropriate matched state in the larger automaton and then intersecting all these resulting automata we get the final commonly shared behavior. Therefore, the final remaining automaton is based on all possible stop position of the smaller automaton and the behavior they may leave in the larger one.

A simple example of the matching of states is portrait in figure 4.5. This examples shows the matching of both normal and exception transitions and the produced result. Nodes 1 and 6 match since the larger automaton (to which the simulation operation was applied to) contains all the smaller transitions. Node 2 unfolds to match both 7 and 8. The matching on exception transitions is the reversed condition of the matching of normal transition, therefore nodes 3 and 9 are matched together since 9 contains all of 3's exceptions. 4 and 10 share similar transitions to a terminal state. Finally, the intersection of all terminal states leads to the resulting stopped automaton (node 13).

We also include the same simulation examples as presented in the formalization section in figures 4.6, 4.7, 4.8, 4.9, 4.10. However, the resulting protocols, although always compatible with the results in that section, might contain some additional behavior as the produced

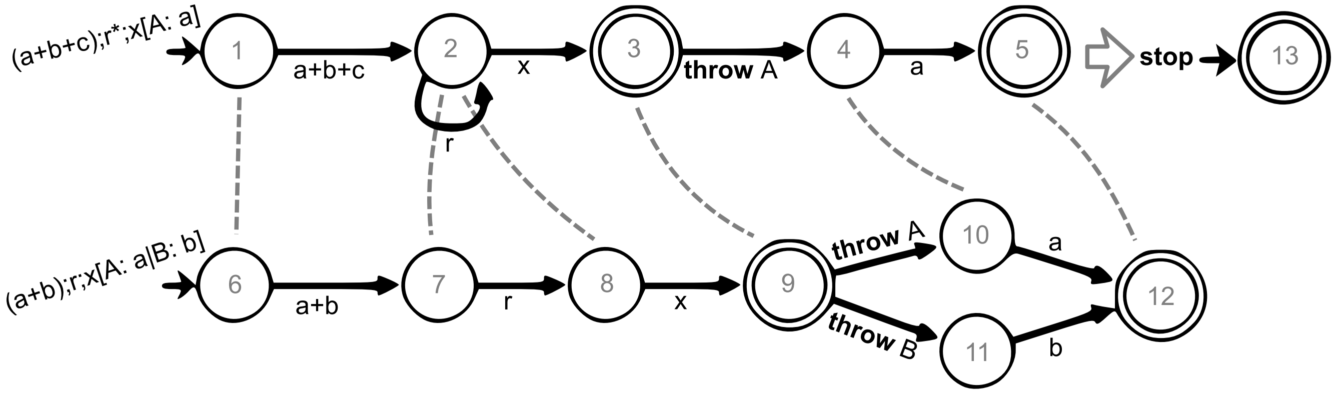


Figure 4.5: $a + b + c; r*; x[A : a] \overset{a+b;r;x[A:a|B:b]}{\longrightarrow}$ **stop**

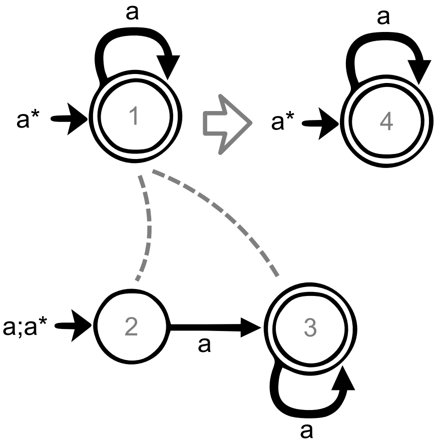


Figure 4.6: $a^* \overset{a;a^*}{\longrightarrow} a^*$

automaton must accommodate all simulation possibilities for those arguments (instead of just checking if a result is valid).

Of these examples, we are only going to briefly describe figures 4.6 and 4.10 as they are the only ones with a resulting automaton. On the first one the matching of nodes is quite simple and the interesting part is the final intersection. For that, node 3 is linked back to node 1 in the original automaton, since it is the only terminal node the final automaton is itself by default. On figure 4.10, after the matching is completed, the automaton $a[I : a]$ contains two end nodes (2 and 4) which are then linked back to nodes 6 and 9 in the larger automaton. By intersecting the two we then obtain an automaton with the protocol b as it is the only shared behavior of the previous ones. This means after using the smaller automaton it is still required for one additional call to b to be made even if the automaton from node 9 does not actually require it as it is needed to satisfy the automaton from node 6.

We will now present a more clean version of the create code. Although refraining from showing all of the simulation algorithm, we instead include the central piece of it: the matching and

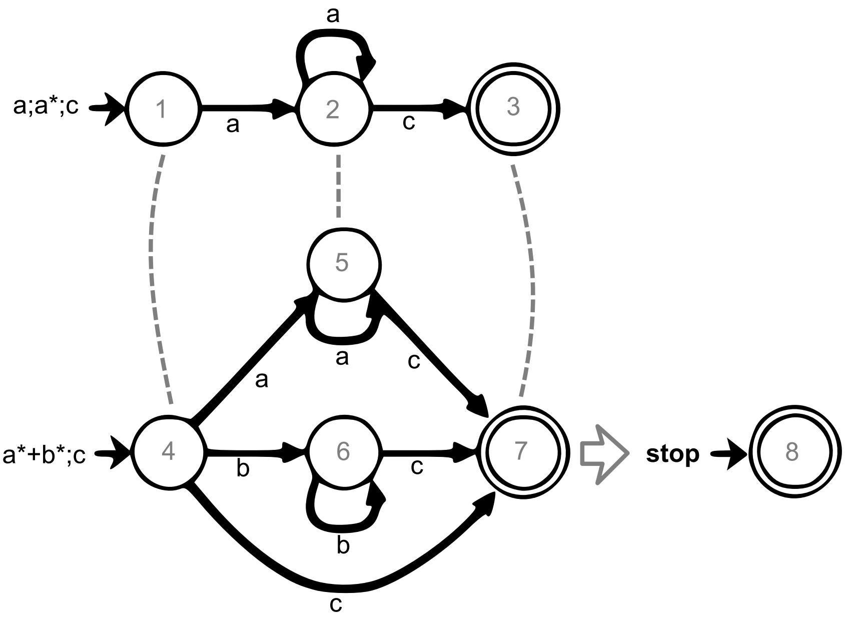


Figure 4.7: $a^* + b^*; c \xrightarrow{a;a^*;c}$ **stop**

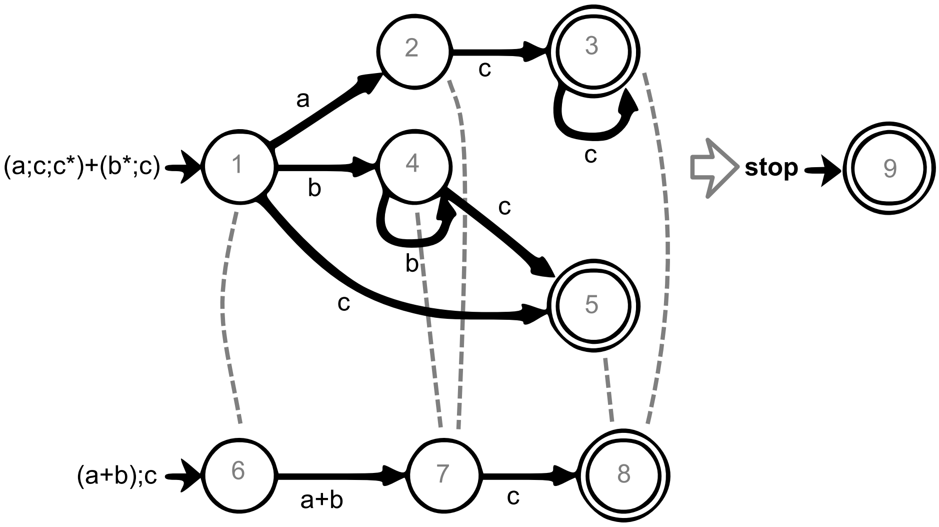


Figure 4.8: $(a; c; c^*) + (b^*; c) \xrightarrow{(a+b);c}$ **stop**

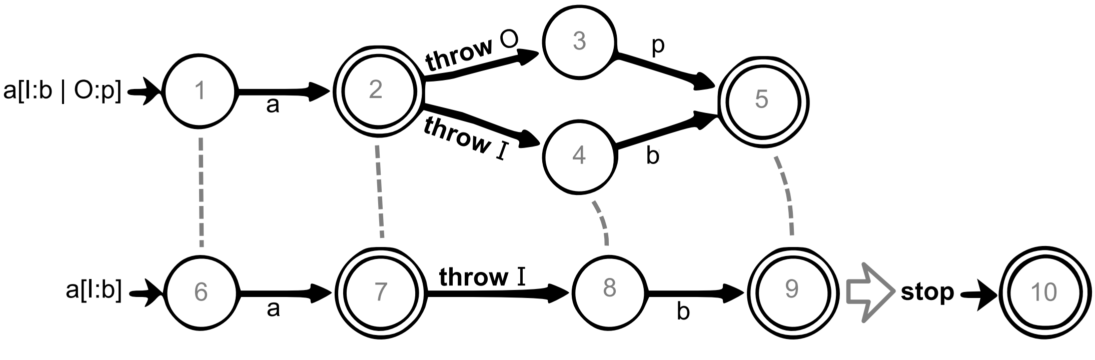


Figure 4.9: $a[I:b] \xrightarrow{a[I:b|O:p]}$ **stop**

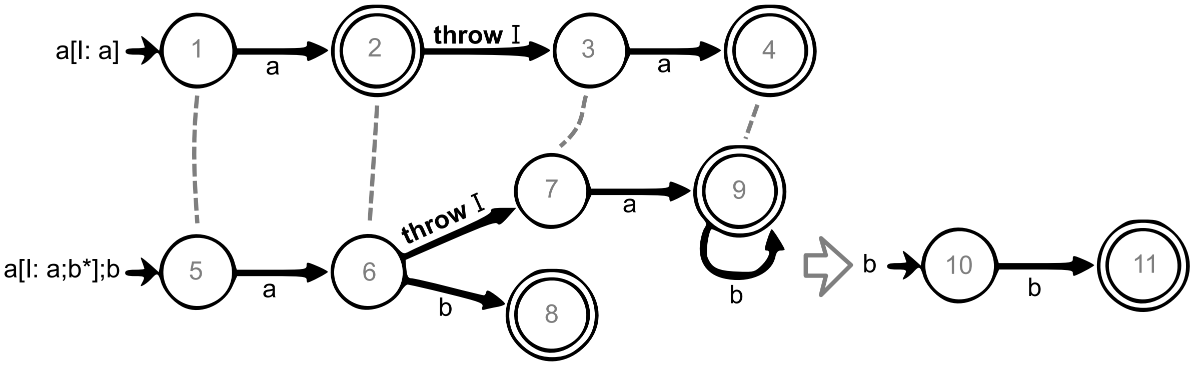


Figure 4.10: $a[I : a; b^*]; b \xrightarrow{a[I:a]} b$

transition-checking algorithms.

Starting by the code that is intended to test if two transition sets are compatible and if so return the pairs of states (nodes of the automaton) which should be matched later.

```
/** pseudo code for "dk/brics/automaton/YakOperations.java:289" */
checkSubsetTransitions( Set<Transition> sub, Set<Transition> sup,
   boolean exceptions_enabled ) {
   Set< Pair<State,State> > result = new Set<Pair<State,State>>();

   //sub must have all non exception transitions of sup
   for( Transition tsub : sub ){
      if( tsub.isExceptionTransition() )
         continue;

      boolean found = false;
      for(Transition tsup: sup){
         if ( tsup.label == tsub.label ) {
            result.add( new Pair<State,State>(tsub.to,tsup.to) );
            found = true;
            break;
         }
      }

      if( !found )
         throw ERROR;
   }

   if( !exceptions_enabled )
      return result;

   //sup must have all exception transitions of sub
   for(Transition tsup : sup){
      if( !tsup.isExceptionTransition() )
         continue;
      boolean found = false;
      for(Transition tsub: sub){
         if ( tsub.label == tsup.label ) {
            result.add(new Pair<State,State>(tsub.to,tsup.to));
            found = true;
            break;
         }
      }

      if( !found )
         throw ERROR;
   }

   return result;
}
```

Complexity wise, this simple method will iterate over both sets of transitions and check if they are included in the previously described way. (Actually, this kind of compatibility check must be done twice: one for the normal transitions and another one for the exception transitions, however they are both in the same (generic) transition set.) Therefore, the overall complexity of this algorithm is $O(n^2)$, where n is the largest number of transitions in either set.

```
/** pseudo code for "dk/brics/automaton/YakOperations.java:226" */
match( State s1, State s2 ) {
   Set< Pair<State,State> > match = new Set<Pair<State, State>>();
   List< Pair<State,State> > work = new List<Pair<State,State>>();
   Pair<State, State> start = new Pair<State, State>(s1,s2);
   work.add( start );

   while( !work.isEmpty() ){
      Pair<State,State> p = work.removeFirst();

      Set<Transition> transitions1 = p.a.getTransitions();
      Set<Transition> transitions2 = p.b.getTransitions();

      //exceptions are disabled on first match
      Set< Pair<State,State> > transitions_match =
         checkSubsetTransitions( transitions1, transitions2, p != start);

      if( transitions_match != null ){
         //deterministic thus only one possible match (or none)
         if( match.add( p ) ){

            for(Pair<State,State> pair : transitions_match){

               State next1 = pair.a;
               State next2 = pair.b;

               boolean loop1 = next1.equals(p.a);
               boolean loop2 = next2.equals(p.b);
               if( loop1 && loop2 ){
                  //transition matching should continue from some
                  //other transitions (if any remaining)
                  continue;
               }

               work.add( new Pair<State, State>(next1,next2) );
            }

         }

      }else
         //not matched, flag error.
         throw ERROR;
   }
   return match;
}
```

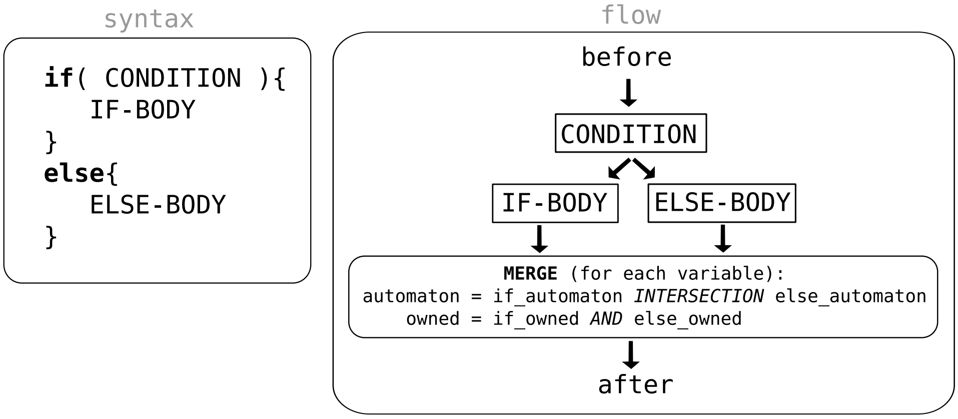
The matching algorithm is simply an iteration over all (possibly) matchable states between the two automata. It avoids repeating already matched pairs of nodes and for each possible pair it might have to do the previously described transition checking. Therefore, this will cost $O(n^2) \times O(n^2) = O(n^4)$, with n being the largest number of either states and transitions.

Finally, it only remains to present the intersection operation that is used directly from the `dk.brics.automaton` library and is used over all resulting automata. Since this operation has quadratic complexity and it has to be applied for each one of those automata, the overall cost will be $O(n^3)$. As such, the complexity of the simulation remains as a $O(n^4)$ operation.

Snapshot of an environment's behavior

When there are different program flows to consider it is needed to store the environment state before testing each branch as the same environment structure is shared between those tests. Therefore, this class is used every time it is needed to save the state of the dynamic type in all currently reachable variables. This is done by copying the content of each one of them in order to keep this information in an unchangeable place. It also allows to copy this content back to an environment, if needed.

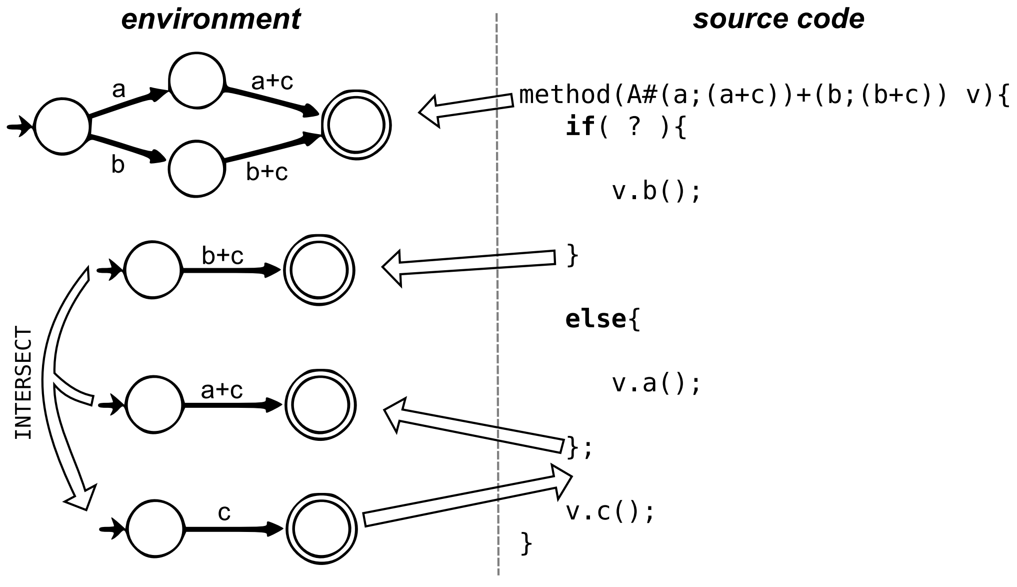
Although this class does just a simple copy ($O(n)$, linear in the number of reachable variable), a more efficient alternative could instead use a kind of "copy on write" method to spare some unneeded duplication when there is unchanged variables. The operation to apply the snapshot to an environment has the same complexity. Note that we only copy the pointers to an automaton since these structures are always shared (an never changed after creation).



```
if( CONDITION ){
   IF-BODY
}
else{
   ELSE-BODY
}
```

```
          MERGE (for each variable):
automaton = if_automaton INTERSECTION else_automaton
    owned = if_owned AND else_owned
```

Figure 4.11: if-else statement syntax and code flow.



```
method(A#(a;(a+c))+(b;(b+c)) v){
   if( ? ){

      v.b();

   }

   else{

      v.a();

   };

   v.c();
}
```

Figure 4.12: if-else example.

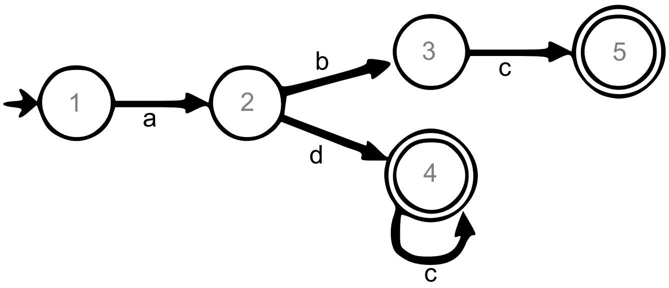


Figure 4.13: `a;((b;c)+(d;c*))` automaton.

4.3.2 Program Constructions

If-Else

This construction introduces a simple branch in the code flow that must be correctly accounted for afterwards. The essential idea is that the normal code flow splits in two and in the end of each branch it must merge back the behaviors into a single commonly shared one (as shown in figure 4.11). Since the behavior is represented by an automaton, this merge is just a simple intersection of each of those two possible different code paths. Additionally, since there is also the ownership flag to take into consideration, the final value of this flag must always avoid possible use errors. Thus, the owned value will only remain true if both sides have it set and, since the disabled flag only restricts the variable's use, there is no additional side effects expected. Intuitively this means that no value can be returned or stored internally if at least one of the branches will not allow it.

Example:

```
method(A#a;((b;c)+(d;c*)) a){
   if( a.a() ){ // A#((b;c)+(d;c*))
      a.b(); // A#c
   }
   else{
      a.d(); // A#c*
   };
   // A#c

   a.c(); // A#stop
}
```

In this example, the type for the variable a starts with the automaton depicted in figure 4.13. Each branch does a different choice that restricts the future allowed behavior by following different paths (node 3 versus node 4). These two protocols have different allowed behaviors and thus they are intersected together which produces a commonly shared behavior that will validate the last call to c.

Finally, the pseudo-code for the implemented solution:

```
/** pseudo code for "src/yak/type/TypeChecker.java:1045" */
visit( ASTIfElse ast , TypeEnvironment env ) {
   //checks if condition is a valid boolean value
   expression( ast.condition(), env, BOOLEAN );

   Snapshot after_condition = new Snapshot(env);
   Snapshot after_if, after_else;

   //if-branch
   boolean if_returns = statement( ast.if_body(), env );
   after_if = new Snapshot(ev);

   //reverts environment to after condition state
   after_condition.copyTo(env);

   //else-branch
   boolean else_returns = statement( ast.else_body(), env );
   after_else = new Snapshot(env);

   //both return -> #stop
   if( if_returns && else_returns ){
      stop( env );
      return;
   }

   //only one returns, the other remains...
   if( if_returns ){
      after_else.copyTo(env);
      return;
   }

   if( else_returns ){
      after_if.copyTo(env);
      return;
   }

   //neither returns, for each variable do
   for( Variable var : env, after_if, after_else ){
      env.var.owned = after_if.var.owned && after_else.var.owned;
      env.var.automaton =
         intersection(
            after_if.var.automaton,
            after_else.var.automaton
            );
   }

}
```

As mentioned before, all typechecking is done based on traveling through each node of the parsed tree and as such, we will comment on this check complexity by ignoring the remaining paths checking complexity. All environment snapshot and copy to operations have a linear computation time and as such the most expensive operation is the intersection of each variable of the two branches at the end of the **if else**. Thus, at this particular node (i.e. ignoring the rest of the code flow in the program) we have a $O(n^3)$ computation time as is given by the automaton intersection operation applied to all reachable variables in the environment.

Note that the **if else** as an expression is very similar to the **if else** as a statement with two main differences: both bodies are evaluated as expressions (instead of as statements) and the final resulting type must be the same in both sides (just like in any normal object oriented type system). Therefore, it will not be presented in this text.

While and repeat

These two loop statements share some similarities as the **repeat** statement is basically a **while** loop with an always true condition. However, this construction is needed since otherwise the typechecker would have to account for possible return values of the condition (not just the type). This would break the typing rules and the general typechecker abstraction layer. It could also lead to some less intuitive implications when the code is changed (and the return value of the condition can no longer be statically known). Thus, the two main differences between these constructions are: **repeat** statements do not have conditions and only the **while** statement needs to continue after the loop (that is, any code after a **repeat** statement is unreachable since there is no break statement in our grammar).

```
i(A# &start( m[Error: start] ) a){
   repeat{
      try{
         a.m();
         return; //only return position
      }
      catch(Error error){
         //loop for another try...
      }
   }
   //this point is never reached
}
```

The previous example shows the specific situation on which the **repeat** statement is necessary. In this case, the protocol specifies that the method m must be run exactly once successfully but if an exception is thrown it must be retried again until it eventually succeeds. For this kind of "retry on error", the **while** statement is insufficient as it assumes there is always the possibility of the condition to fail and on that situation it would not cause the protocol to be completely satisfied.

However, this expressiveness is not needed if recursion is used. This will be detailed in a more appropriate section (see section 4.3.3) but, as is general knowledge, recursive methods have some limitations in regards to stack overflows and overhead.

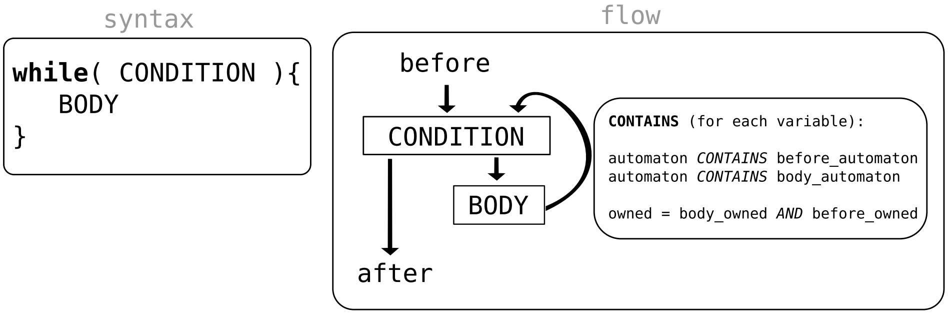

syntax
while(CONDITION){
BODY
}
flow
before
CONDITION
BODY
after
CONTAINS (for each variable):
automaton CONTAINS before_automaton
automaton CONTAINS body_automaton
owned = body_owned AND before_owned

Figure 4.14: while statement syntax and code flow.

```
r(A# &start( m[Error: start] ) a){
   try{
      a.m();
      return; //recursion base
   }
   catch(Error error){
      r(a); //recursion loop
   }
}
```

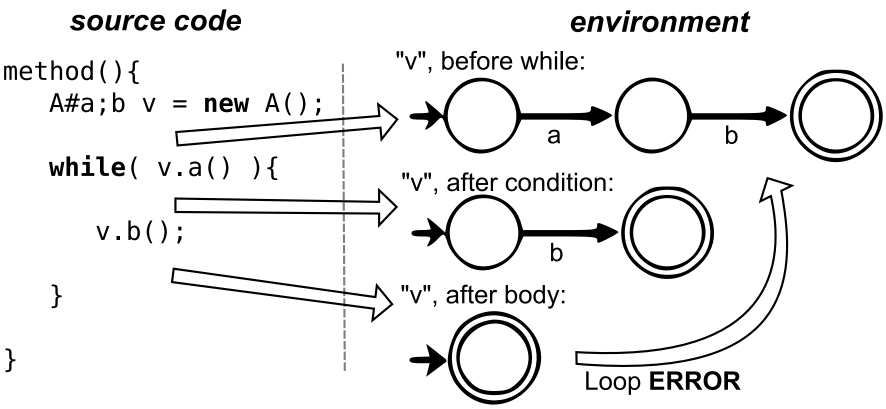
As such, both loops will be presented in this same section with the **repeat** being a special case of the more generic **while** loop.

The **while** flow is a type of code block that can be repeated zero or an unpredictable number of times and that will always end after an evaluation of its condition. In order to allow another evaluation of that condition it is required for the state of the environment after the evaluation of the loop's body to be compatible with the state before the loop is called as shown on figure 4.14.

However, there are three different possible situations for the state of the automaton after an evaluation of the body: it could be in a more generic state that allows "more" behavior, it could be in a more restrictive state or it could even be in an incompatible state. The last situation is clearly incompatible with this statement flow and therefore is flagged as an error (see figure 4.15) but the other previous two must be individually analyzed.

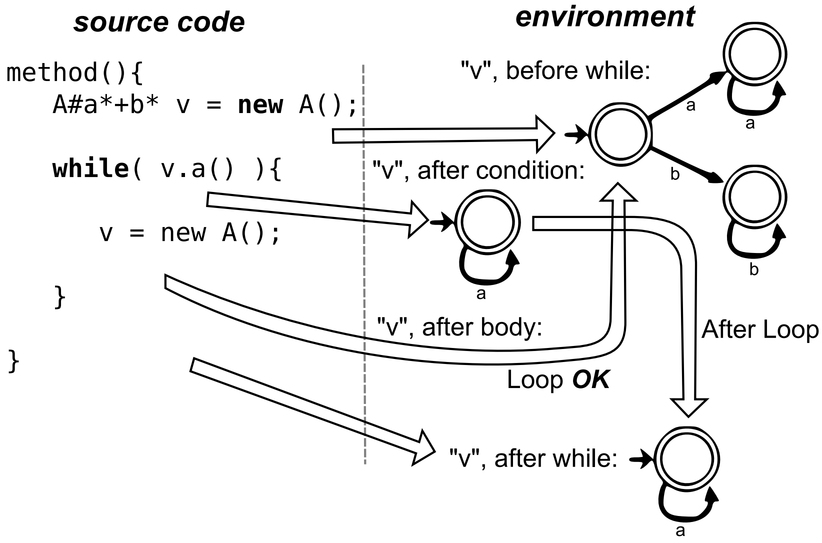
In the first case, when the behavior is more generic (figure 4.16), this will never cause problems since to reach this state all the code in the **while** was already checked to be compatible with a more particular situation (smaller automaton) and therefore there is no special additional check needed and the flow may continue normally with the behavior after the condition.

In the second case (figure 4.17), however, the situation is a little more complicated. It is unclear if the more restrictive behavior can be used or not in a possible next loop. This situation can happen when the **while** body does a choice in the automaton that will influence the allowed future behavior (after the loop). In order to propagate the choices and know if this restriction is compatible with the loop, the check is repeated with this new behavior as if it were the initial one. This should eventually stop since the group of possible sub automata in a deterministic finite automaton is limited and there can not be any oscillation since it is always



```
method(){
   A#a;b v = new A();

   while( v.a() ){

      v.b();

   }

}
```

Figure 4.15: While statement with incompatible flow.



```
method(){
   A#a*+b* v = new A();

   while( v.a() ){

      v = new A();

   }

}
```

Figure 4.16: While statement where the body leaves a more "generic" behavior.

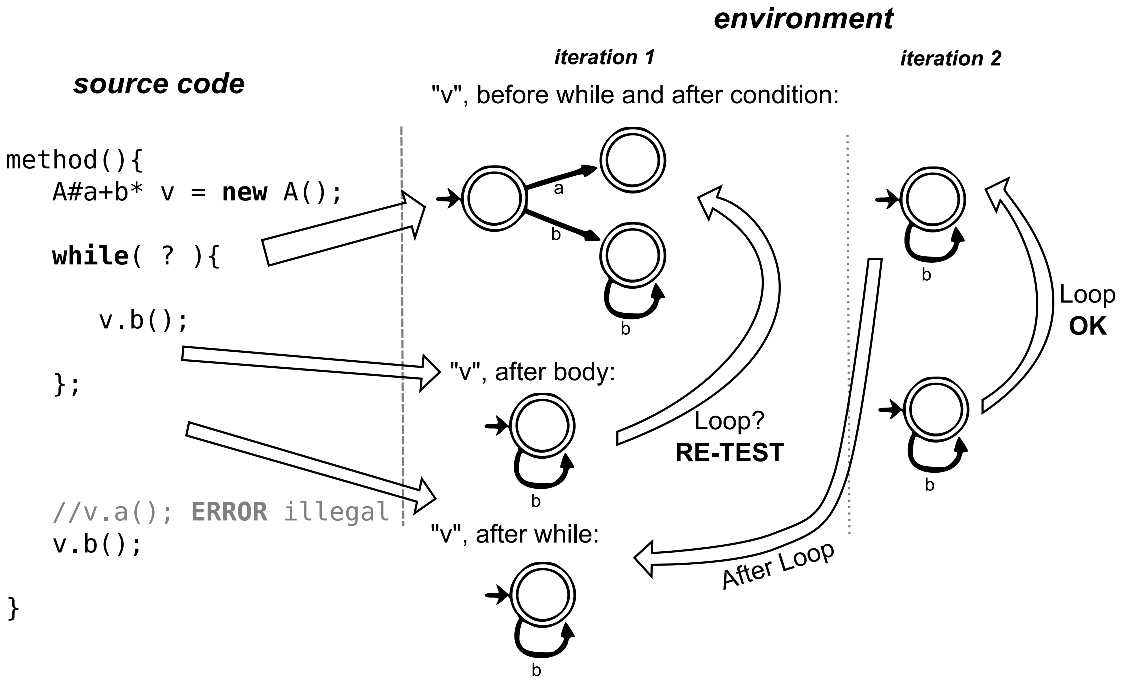


Figure 4.17: While statement where the body leaves a more "restrictive" behavior.

required the old one to be fully contained in the start automaton of the next retry. However this could potentially require to travel though all possible automaton's state before a conclusion is reached (as each state may form the start node of a sub-automaton - note that this is mostly related to option nodes (the choice and exceptions) not sequential constructions).

Thus, the implemented solution must account for all these three possibilities and check the program code accordingly. For the **repeat** statement there is neither a condition nor an after loop environment, and therefore it is just required for it to be able to loop back to the starting point of the repetition with the same kind of possible restrictions as explained before.

Pseudo-code of the implementation:

```
/** pseudo code for "src/yak/type/TypeChecker.java:906" */
visit(ASTWhile w, TypeEnvironment env) {

   //needs another go to check the loop...
   boolean again = true;

   while(again){
      Snapshot before = new Snapshot(env);

      //repeat does not have a condition
      if( !w.isRepeat() )
         condition( expressionDiscardCheck(env,w.condition()) );

      Snapshot condition = new Snapshot(env);
      boolean body_returns = body(env,w.while_body());
      Snapshot body = new Snapshot(env);
      //body throws exception or returns (no loop)
      if( body_returns ){
         if( w.isRepeat() )
            stop( env );
         else
            condition.copyTo(env);
         return;
      }

      again = false;

      for( Variable var : env, condition, body ){
         env.var.owned = condition.var.owned && body.var.owned;

         if( body.var.hasSimulation(before.var.automaton) ){
            //body is "larger"
            env.var.automaton = condition.var.automaton;
         }
         else {
            //body is "smaller"
            if( before.var.automaton.hasSimulation( body.var.automaton ) ){
               env.var.automaton = body.var.automaton;
               again = true;
            }
            else throw ERROR;
         }
      }
   }
}
```

The implementation actually groups both **while** and **repeat** AST nodes together in the same visit method. The distinction is only made by a simple *isRepeat* method. The algorithm starts by checking both the condition and the body of the loops and with those states saved it then decides if it needs to recheck this statement with a smaller automaton by checking the behaviors of each of the reachable variables.

- *body* is more generic: this means the automaton from the body environment has a larger behavior, and therefore also contains the behavior of the environment before the loop is called. In this case the variable will not require another test.

- *body* is contained in the before automaton: in this case, the more restrictive behavior has to be validated with the loop's body. This means it is needed to repeat the check with the initial environment in as if this smaller automaton were the starting state for the loop.

- *body* neither smaller nor larger: this situation is flagged as an error, since it is impossible for the behavior to legally allow another loop iteration.

If we ignore the complexity of checking the body and condition, then this verification will have $O(n^6)$ time (although two simulations may occur per reachable variable) since the basic loop check operation ($O(n^5)$) can be repeated at most the length of the largest automaton in a variable.

Assignment

In the prototype, an assignment operation is handled in a much simpler way as we make extensive use of pointers. This means that instead of having to control each variable's dynamic type explicitly it can be done just over the dynamic type structure as its real location is not important. This leads to some code simplification and more freedom on using a behavioral type in some situations without requiring it to be stored in a variable. Another difference to the type system rules is that we allow for the prototype to accept the null value on any stack variable, independent of the static type.

The main checks of this construction will now be introduced in the same sequence as they appear in the pseudo-code.

- the dynamic type of the variable must be in an accept state, as it is behavioral view will be lost after the assignment of the new value;
- **`null`** values are only allowed when on stack variables (and will be seen as stopped types);
- the new type must be a valid subtype of the declared static type of the variable;
- if the static type is just a **`stop`** protocol, there is no need to change the possession of the new type as this behavioral view does not interfere with it. Therefore, all ownership values remain unchanged;
- the ownership flag must be set in the new type (that the variable receives) if the static type requires it;
- the new type must reach a valid end state after being used with the static type's behavior. This is modeled as the simulated behavior of the static protocol must fulfill the new type's behavior;
- Finally, the ownership needs to change from the new type to the new content (which will be stored in the variable). This means the new type will become stopped and the variable receives the clone (since we are handling pointers we can not simply return the new type).

```
/** pseudo code for "src/yak/type/TypeChecker.java:1239" */
assign(Value static_t, Value dynamic_t, Value new_t, boolean stack){
   if( dynamic_t != null && !dynamic_t.getProtocol().isAccept() )
      throw ERROR;

   //null handling
   if( new_t == null ){
      if( !stack && !static_t.getProtocol().isStop() )
```

```
         throw ERROR;

      return static_t.stoppedClone();
   }

   subtyper.check(static_t,new_t);

   if( static_t.getProtocol().isStop() )
      return static_t.stoppedClone();

   if( static_t.getOwned() && !new_t.getOwned() )
      throw ERROR;

   if( !new_t.getProtocol().hasSimulation( static_t.getProtocol() ) )
      throw ERROR;

   new_dynamic_t = static_t.clone();
   new_dynamic_t.setOwned( new_t.getOwned() );
   new_t.setStopProtocol();

   return new_dynamic_t;
}
```

The only real complex code is the simulation operation and as such it pushes the whole assignment check to the expected $O(n^4)$ computation time.

Method call and argument packing

Before a method call can take place, we must first check if the arguments are compatible with the method's signature. This starts by checking the resulting type of the type evaluation of each argument expression. These types are then checked for uniqueness by testing the produced set on all non stop parameters (not shown in the pseudo-code).

After that, each method must then be checked if the dynamic type is compatible with the static type of the argument variable. Since this is modeling a method call it must also change the type's behavior to the after-call state, as if the call had been made and was already completed.

Pseudo-code, line by line:

- if the protocol is non stop, it can not receive a `null` value as that specific argument;
- the expected received type must be compatible with the one currently under check;
- non stop parameters will not need to be forwarded as their behavior remains the same;
- the protocol of the dynamic type must correctly simulate the protocol of the static type so that all the static behavior is contained in it;
- the ownership must be set if it is required. Since this means the type is transferred to the method's body, the argument's protocol must complete the dynamic type in order for it to reach an acceptable state. After this, the dynamic type can not be further used and so it

is forced into a stopped protocol (the accept state only requires for it to be safely stopped at that point, but some additional behavior might also remain available and therefore it must be explicitly blocked).

```
/** pseudo code for "src/yak/type/TypeChecker.java:1162" */
argument(Value static_t, Value dynamic_t) {
   if( dynamic_t == null && !static_t.getProtocol().isStop() )
      throw ERROR;

   subtyper.check(static_t,dynamic_t);

   if( !static_t.getProtocol().isStop() ){
      if( !dynamic_t.getProtocol().doSimulation( static_t.getProtocol() ) )
         throw ERROR;

      if( static_t.getOwned() ) {
         if( !dynamic_t.getOwned() )
            throw ERROR;

         if( !dynamic_t.getProtocol().isAccept() )
            throw ERROR;

         dynamic_t.setStopProtocol();
      }
   }
}
```

Again, this particular code block suffers from the complexity of the simulation operation and since the overall check will apply this operation to each one of the arguments, the total time will be $O(n^5)$.

We will not show in here the code that checks an expression give as argument and also if that set has no (behavioral) repetitions, since we use pointers the checking is trivial. Thus, the following code just handles a simple method call with possible call throws (each of them with a logic that will only be introduced in section 4.3.2 - **`throw`** statement - as they share the same code).

```
/** pseudo code for "src/yak/type/TypeChecker.java:1334" */
call(TypeEnvironment env, Value caller,String method,Value...args){
   if( !caller.hasMethod(method, args) )
      throw ERROR;

   MethodValue m = caller.method(method, args);
   Value result = m.check(caller, args);

   if( env.context().isInternalCall(caller, m)){
      Snapshot before_call = new Snapshot().snapshot(env);

      env.context().checkInternalCall(this,context,m);
      Snapshot after_call = new Snapshot().snapshot(env);

      for(Value throwed : m.getThrowns()){
         before_call.copyTo(env);
         thrownException(m,env,caller,throwed);
      }

      after_call.copyTo(env);

   }else{
      for(Value throwed : m.getThrowns())
         thrownException(m,env,caller,throwed);
   }

   discardWatcherAdd(result);
   return result;
}
```

This also shows a change in the prototype in relation to the check of a possibly discarded value. Instead of always forcing a value to be stored inside a variable we allow for it to be more freely used. Therefore, we must save the pointer to each of those types and assure it was either completely consumed in that statement (so it may be safely discarded) or that it is now saved somewhere so it may still be used later.

Return

Returning a value has some interesting additional constraints that must be respected in order to control the overall behavior of the application.

First, it is needed to check the behavioral permission of returning the type. This is simply the same condition as used before on the assignment with the exception that there is no previous content in the variable (thus it is **null**) and it is not allowed for returning a **null** value as representative of some other non stopped behavioral type. By using the assignment rule we also check automatically the subtyping validity of the result and some additional side-effects are immediately applied (like removing the ownership of the returned object).

Each existing **return** statement in the same method must be coherent among each other. Therefore, we must also check the intersection of the field's behavior with possible previous **return** statements. This operation stores the shared behavior among these cases so that it may be used later in the consistency check.

Lastly, all local environment variables must be in an acceptable terminal state as they will fall out of scope and so no behavior can be left incomplete.

This operation carries the complexity of the assignment as everything else done in here has less weight.

```
/** pseudo code for "src/yak/type/util/Context.java:98" */
checkReturn(Value returned,TypeEnvironment return_environment){
   assign(method.getReturnType(),null,returned_type,false);
   //checkEnvironmentEnd(return_environment);
   for( Variable var : return_environment ){
      if( !var.dynamic_type.isStop() )
         throw ERROR;
   }

   //intersectFields();
   for( Variable var : fields, return_snapshot){
      return_snapshot.var.automaton.intersect(fields.var.automaton);
   }
}
```

Throw

The **throw** statement is handled in a very similar way as a **return** with the main difference being that it has to end all environments up until it reaches the **try catch** construction (or the **throws** declaration). It also has to save the common behavior of the throw environment with the one that was thrown by some other previous **throw** statement of the same type. As usual, this is simply done by intersecting the environment with the catch environment for that type.

```
/** pseudo code for "src/yak/type/util/Context.java:186" */
catchesException(Method method, Value thrower, Value throwed,TypeEnvironment env){
   SnapshotAndTryEnv previous = exceptions.find(throwed);

   if( !thrower.getProtocol().isStop() ){
      if( thrower != self ){
         Value clone = thrower.automaton.clone();

         if( clone.doThrowTransition(throwed) ){
            env.replace(thrower, clone);
            //checkEnvironmentEnd( env, previous.try_env );
            for( Variable var : rnv < previous.try_env ){
               if( !var.dynamic_type.isStop() )
                  throw ERROR;
            }

            if( previous.nonBehavioral() )
               checkReturn( getReturnType(), env ):
            else
               found.caught.intersect(env, found.limit);

            env.replace(clone, thrower);
            return;
         }
      }else{
         CacheNode cache = new CacheNode(method,throwed,this);
         MapStruct s = env.context.throw_cache.get( cache );
         s.caught.copyTo(env);
```

```
            }
        }
    }

    //checkEnvironmentEnd( env, previous.try_env );
    for( Variable var : env < previous.try_env ){
        if( !var.dynamic_type.isStop() )
            throw ERROR;
    }

    if( previous.nonBehavioral() )
        checkReturn( getReturnType(), env ):
    else
        found.caught.intersect(env, found.limit);
}
```

After the throw type is obtained, we must register and check the compatibility of the environment's state with any other of the same type that may have been thrown before (simple intersection, again). For that we first get the structure that stores the snapshot of the environment and the top environment on which the exception is caught.

The handling of a behavioral exception (that causes a protocol change after the **`throw`**) is done in a similar way to a normal exception. The main difference is in the changing of the environment's state for that particular variable's dynamic type and then reverting it to the normal path before returning.

On an internal call (since it will only appear in the last recursion pass - see section 4.3.3) the state of the fields is always known, thus it is only needed to copy that snapshot back to the environment before continuing to the normal **`throw`** check.

For the **`throw`** to be correct we must then check if the variables which will fall out of scope (an thus that are not on the environment at the **`try catch`** level), are safely stoppable. After that, the only remaining action needed is to save the environment behavior for a possible **`catch`** branch. However, if the exception is in the method's signature of a non behavioral exception then there is no need to account for the **`catch`** environment has they can not change the fields behavior anyway.

From a spatial complexity point of view, each **`catch`** has to store a single copy of all reachable environment variables' behavior even if an arbitrarily large number of intersections are applied to it.

Try-catch

In order to check a **`catch`** it is required to capture all possible behaviors at any **`throw`** point of the respective type that may occur inside the **`try`** body. Thus, all the environments in which a **`throw`** of that type may happen must be compatible among themselves. This is necessary since all those positions in the code may jump to a shared **`catch`** which must account for all those possible situations.

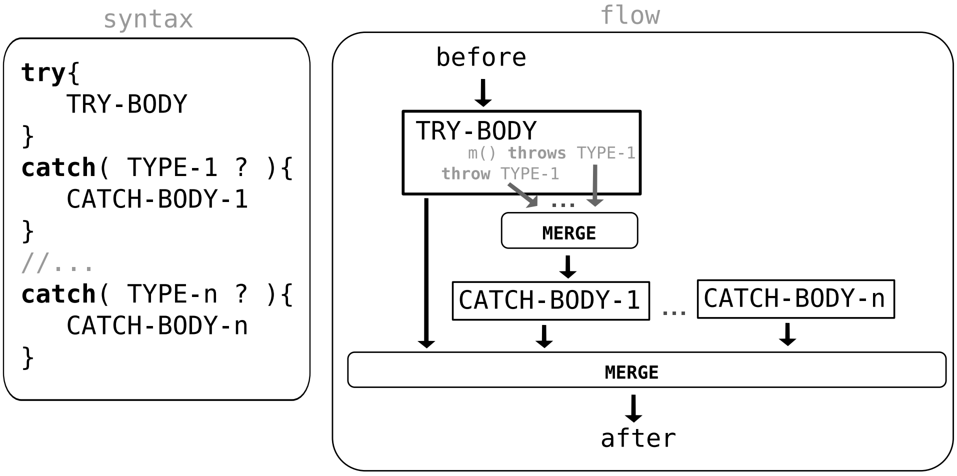


Figure 4.18: try-catch statement syntax and code flow.

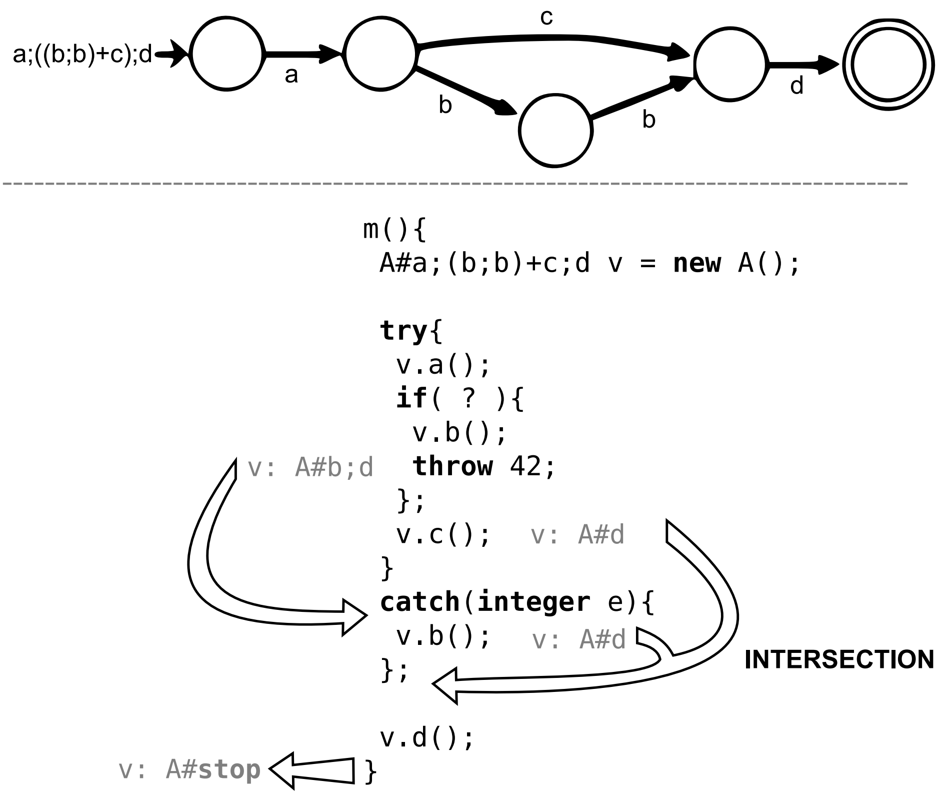
Note that these throws might occur not only explicitly (with the **throw** statement) or implicitly (when a method declares a **throws** list - each one of those is treated as a single **throw** statement). Finally, the threw environment is captured after any possible behavioral exception transitions. This means that after a **throw** the behavior is automatically changed to the respective "exceptional/special behavioral". It is also relevant to remember that all behaviors in the environments that will lose scope must be in an acceptable state so that they can be safely discarded.

This statement has a less obvious flow since any throw/call-throw statement inside the **try** branch will cause a jump to the appropriate **catch**. Therefore, there can be one or more positions in the **try** code for this situation to happen but they are all merged together as described previously. As such, this only leaves two checks remaining: the validation of the **catch** body with the threw environment and obtaining the environment at the end of this statement. The resulting environment is simply the combination of all possible end behaviors (catch branches and normal flow) and as usual this is done with the simple intersection of those environments to determine a compatible and common behavior (if it exists).

```
method(){
   A#a+(b;c);c v = new A();

   try{ //A#a+(b;c);c
      if( ? ){
         v.a(); //A#c
         throw 0;
      };

      v.b(); //A#c;c
      throw "flag";
   }
   catch(integer i){
      //A#c
   }
   catch(string s){
      //A#c;c
      v.c();
      //A#c
   };
   //after try A#c
   v.c();
}
```

The pseudo-code:

```
/** pseudo code for "src/yak/type/TypeChecker.java:360" */
visit(ASTTryCatch tc, TypeEnvironment env) {
   Context context = env.context();
   context.newExceptionHandlerLevel();

   //register each catch branch
   for( ASTCatch c : tc.catches() ){
      context.addExceptionHandler( c.type_name() );
   }

   //try-body
   boolean returns = body( env, tc.try_branch() );

   Map<String,TypedEnvironment> exceptions = context.endExceptionHandlerLevel();

   Snapshot after = null;

   if( returns )
      after = new Snapshot(env);

   //check each catch body
   for( ASTCatch c : tc.catches() ){
      TypeEnvironment alter_throw = exceptions.get( c.type_name() );

      after_throw.copyTo( env + c.parameter() );

      boolean returns = body( env, c.body() );

      if( returns )
         continue;

      if( after == null ) {
         after = new Snapshot(env);
         continue;
      }

      //normal intersection of environments
      after.intersect(env);
   }
```



```
m(){
 A#a;(b;b)+c;d v = new A();

 try{
  v.a();
  if( ? ){
   v.b();
   throw 42;
  };
  v.c();
 }
 catch(integer e){
  v.b();
 };

 v.d();
}
```

Figure 4.19: Simple try-catch example.

```
   //all returned (no after)
   if( after == null ){
      stop( env );
      return;
   }

   after.copyTo(env);
}
```

In figure 4.19 we show how the **try catch** flow is handled for a single stack variable.

4.3.3 Checking class consistency

In order to check the body of a behavioral method it is needed to take into account its behavioral context on which the call will be made. Since the class' fields change their behavior in accordance to the sequence of calls made, it is not possible to determine if a method body is correctly constructed without also considering the fields expected behavior on the moment of a particular call. Therefore, we need to do this kind of class consistency check to verify if, given the class usage protocol, all uses of the class' fields in any behavioral method body respect the field's declared behavior.

Any class field has its use split among possible several method bodies. This may raise problems when those variables have non stop behaviors that must be accounted for. Since methods whose name is not contained in the protocol can be freely called and thus cause unpredictable interferences, those variables are limited to a "constant view" in the sense they can not assign new values or change the field's behavior (by calling behavioral methods, etc). Therefore, only behavior methods whose context is always known (it is declared in the protocol) can modify behavioral fields.

However, this kind of verification is made independent of the context for any non behavioral method check or if the usage protocol is **stop** (thus it does not have any behavioral methods). In this last case since there will never be any behavioral interferences from different uses of the class' fields. As such, for that case, we allow any method to use the fields by forcing them to have stopped behaviors in a similar fashion to normal static variables (but only visible at the class level).

```
/** pseudo code for "src/yak/type/util/ConsistencyChecker.java:74" */
checkConsistency(ClassValue self){
   Collection<String> fields = self.getBehavioralVariablesNames();
   Context context = new Context(self);

   //1st - check non behavioral methods
   boolean stop = self.getProtocol().isStop();

   //block all behavioral variables
   for(String s : fields){
      VariableValue variable = self.field(s);
      variable.dynamic_type.setStopProtocol();
      variable.setLock( !stop );
   }

   for(MethodValue method : self.getBehavioralessMethods()){
      context.clearFields();
      context.resetStack();

      //check method body
      checker.checkMethod(method,context);

      //end node, class variables must be in accept state
      if( stop )
         checkFieldsEndOfPath(context.getFields());
   }

   //2nd - don not check path stopped or no fields
   if( stop || fields.size() == 0 ){

      //normal method check
      for(MethodValue method : self.getBehavioralMethods() ){
         context.clearFields();
         context.resetStack();

         checker.checkMethod(method,context);
      }
      return;
   }

   //behavioral methods check
   //unlock all state variables (but keep null-state)
   for(String s : fields){
      Variable variable = self.field(s);
      variable.setLock(false);
   }

   checkMethodsConsistency();

}
```

Before starting any check, all fields dynamic type are set to **stop** (**null**). As said before, variable locking is only made if the usage protocol is non stop. We then proceed to check each behavioraless method. There is also no need for contextual method check when there are no field variables since interferences will never happen. Finally, this code ends with the unlock of the fields and the call to a (soon to be described) method that checks the consistency of the method's body with the context given by the declared protocol.

When a class declares a non stop usage protocol the methods in that expression must be used according to a specific context in the sense that they must obey a concrete sequence of calls (traces). This gives all the information needed to verify the correctness of the fields behavior. Basically this is simply done by traveling through the protocol's automaton while controlling all fields behaviors at the beginning and ending of each labeled transition.

This kind of check also assures the correct code modularity (each class must contain a correct block that can then be combined with others - also correct - ones) and that each class respected the declared typing annotations.

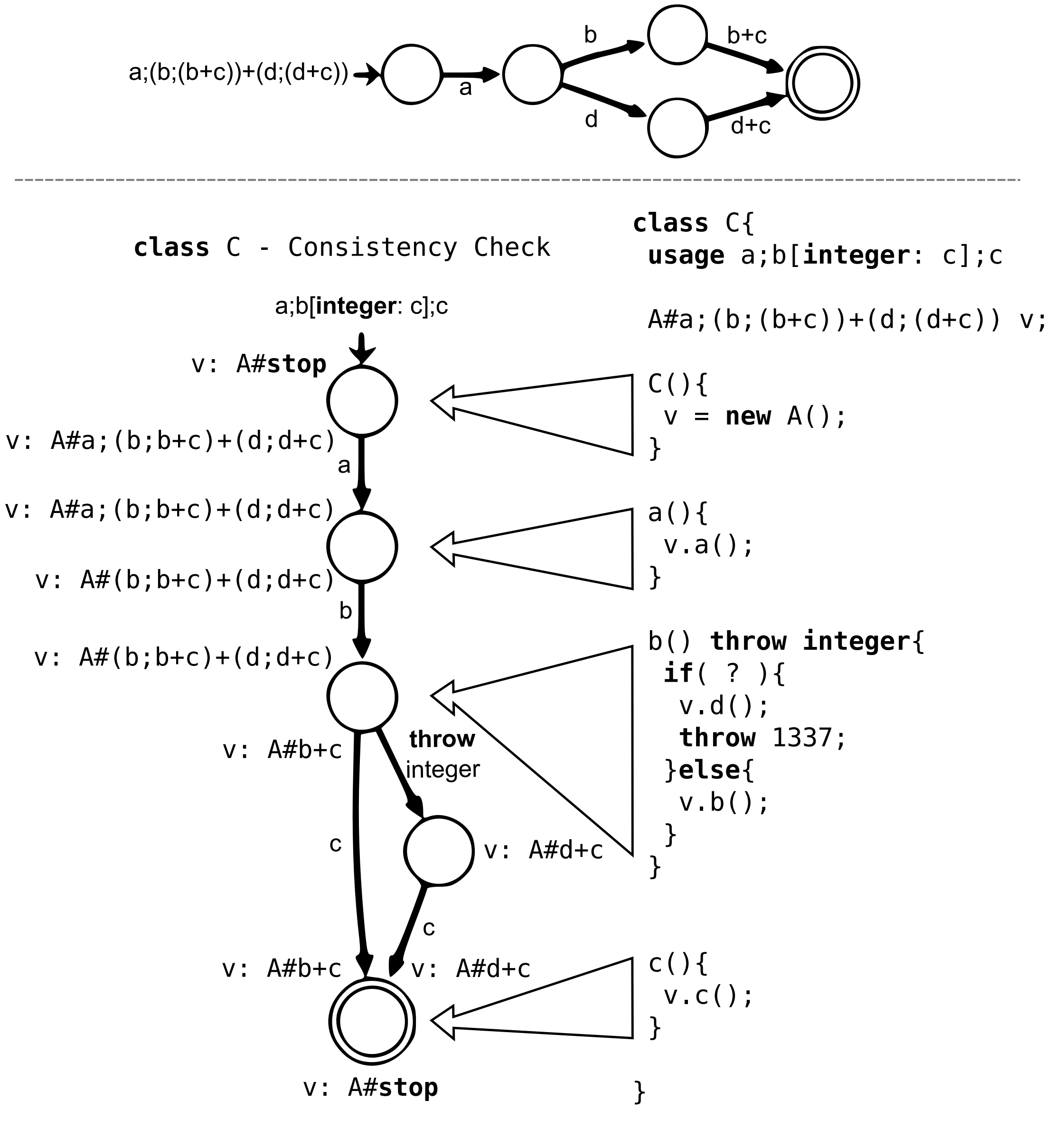
The core of the algorithm is simply a depth first search for all possible conflicting behaviors. We also warn about unused paths since they can not be tested.

```
/** pseudo code for "src/yak/type/util/ConsistencyChecker.java:156" */
checkMethods(){
   List<Node> work = new List<Node>();
   work.addFirst( new Node(
      constructor(), //method to check
      initialState(), //state
      new FieldsSnapshot(self,"#stop") )  //fields' state
   );

   visited = new Set<StateArray>();

   while( !work.isEmpty() )
      checkMethod( work.removeFirst(), work );
}

/** pseudo code for "src/yak/type/util/ConsistencyChecker.java:177" */
```

```
checkMethod( Node node, List<Node> work ){
   FieldsSnapshot out = checkSameNameMethods(node,rc);

   //if this node is terminal (end of a path)
   //all class fields must be in accept state
   //useless -> if never returns
   if( !useless(out) && node.state.isAccept() )
      checkFieldsEndOfPath(out);

   for( Transition t : node.state.getTransitions() ){
      //true if not already visited
      if( visited.add( new StateArray(node.state,t.getDestination(),out) ) ){
         FieldsSnapshot in = out;

         //switches to the appropriate exception FieldsSnapshot
         if( t.isException() )
            in = exception(out);

         if( useless(in) )
            throw ERROR;

         work.addFirst( new Node(t.getName(),t.getDestionation(),in) );
      }
   }
}

/** pseudo code for "src/yak/type/util/ConsistencyChecker.java:232" */
checkSameNamedMethods(Node node){
   //consistency for same named methods (like: A/0, A/1, etc.)
   for(MethodValue m : self.nameMethods(node.method) ){
      //revert class variables to before-call state
      context.setFields(node.snapshot);
      checker.checkMethod(m,context);
   }

   return context.getFields();
}
```

We start check by pushing the constructor methods and the initial automaton state into the work stack, starting from an all stop state on each field. For each element in that stack we then check the body of the method and add to the stack all possible (not visited) transitions that may follow. Note that the visited condition also considers the state on each field. The fields state is the returning one on normal transitions and the respective exception throw state on each of those throws types. The search must also check the correct termination of the field's use on each possible automaton stop position. Finally, it is also important to note that methods with the same name are checked together. This means that they must have consistent **return** and **throw** field states.

Figure 4.20 shows a simple example of this verification for a simple class.



```
class C{
 usage a;b[integer: c];c

 A#a;(b;(b+c))+(d;(d+c)) v;

 C(){
  v = new A();
 }

 a(){
  v.a();
 }

 b() throw integer{
  if( ? ){
   v.d();
   throw 1337;
  }else{
   v.b();
  }
 }

 c(){
  v.c();
 }

}
```

Figure 4.20: Example of a class consistency check.

The complexity of this operation goes beyond the depth-first search algorithm as it may continue the search even after visiting all nodes in the automaton as well as the price payed for checking the method's body one each node position.

Internal call

We consider the protocol to be only related to the object's external behavior. This means that someone using that object must obey the specific sequence of calls required by its description. However, internally, the object is free to call its own methods in any order it may require. This should not cause any conflicts as long as two special rules are applied: the object's protocol only transitions with external calls (not internal) and non behavioral methods (those who can be freely used in any situation) can not change (non stop) field variable. This last rule is necessary to remove any possible interferences that they may cause since their call context is always unpredictable. As a side-effect, this causes all non behavioral methods to be able to call behavioral one as long as their bodies do not cause any changes in the fields of the class.

In view of that, any internal call although allowed, must be coherent with the current call context (i.e. the fields state). This is done by importing the code to the current method body (figure 4.21). Actually, to avoid too many checks it is used a cache of the internal calls based on the field's state at a particular call point. In other words, if the class' fields are in the same state as in a previously calculated internal call then (since they will cause the same transition as before) the state afterwards is the same as the one in the cache. The base case re-checks the methods body to find the reached state after that particular call has ended. We must also consider not only the normal method termination (on a **`return`**) but also the case when one of the declared exception (in the method's signature) is thrown.

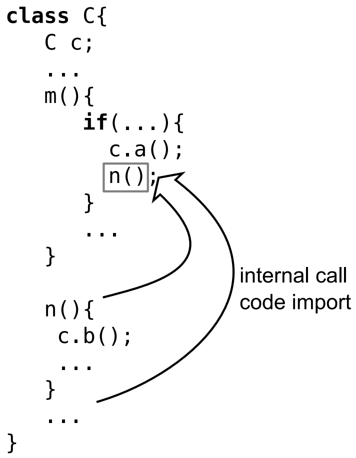
Presenting the pseudo-code for this section is rather complicated as this feature is combined with the recursion algorithm since they have some dependencies between them. Therefore, the following section will also contain some of the logic of what was described in here. The main idea is that, to simulate this code import, we use a kind of fake call stack that tracks the state of the check. Together with the recursion handling algorithm, this will avoid possible loop sequences as the number of possible combination is limited by the automaton.

Call recursion

Recursive calls happen when a method calls itself in its body. As a consequence of the previous section (internal calls), this might happen directly or indirectly when the body imports some other method that in turn will call back one of those which was called before.

The main problem of recursive calls is then to determine the resulting behavior of each class field after the call expression ends. In order to simplify the overall logic, we do this in three different passes through the code:

- in the first (pre-pass), the recursion is removed from the AST tree so that any recursive call is hidden;



```
class C{
   C c;
   ...
   m(){
      if(...){
         c.a();
         n();
      }
      ...
   }

   n(){
    c.b();
    ...
   }
   ...
}
```

Figure 4.21: The code of n will be imported to method m for the type checking.

- on the second (base-pass), we use the previously obtained AST tree to calculate the resulting base behavior for the recursion;
- on the third (full-pass), we just check that the complete (with recursion) AST tree is compatible with the previous behavior obtained in the base-pass. For non recursive method, this is the base-pass.

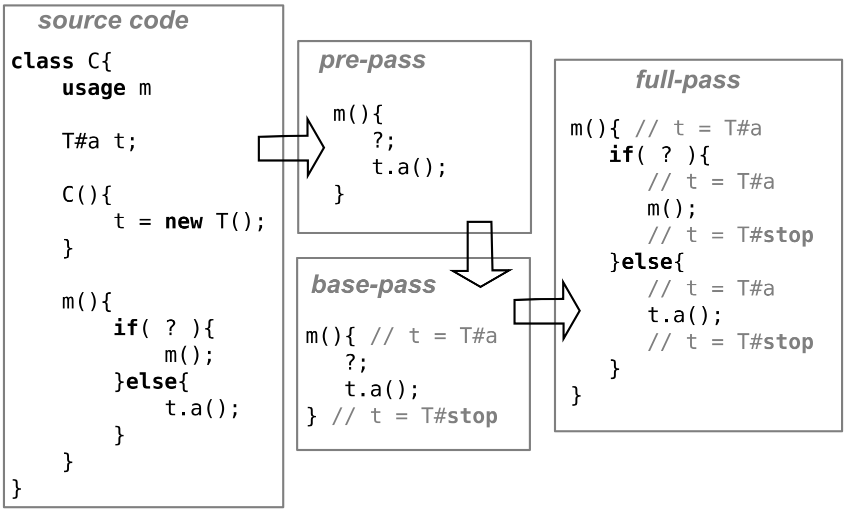
This algorithm basically prunes all of the recursive branches in the code's path which will allow for a kind of pre-evaluation of the method's code. The code to remove the recursion will not be presented as it is too specific to our AST constructions, but it can be seen as simply cutting out all code cases that are not the base one. This should make it easy to calculate the state of the fields when the recursion stops (i.e., at the base case). There is never ambiguity to know if a call is or not internal since due to the behavioral linearity principle applied to the **this** pointer, any internal call has either no prefix or the **this** one. The only possible unknown recursion points derive from non-behavioral calls and all these are irrelevant (to this analysis) since they can not change a field's behavior.

```
/** pseudo code for "src/yak/type/TypeChecker.java:1385" */
checkMethod( ASTMethodValue method, Context c ){
   TypeEnvironment env;

   env = makeEnv(c,method);
   c.setMethod(method,env,base-pass);

   if( isRecursive(method) ){
      //pre-pass
      AST non_recursive = removeRecursion( method, c.methodInStack() );

      c.saveFields();
      statement(env,non_recursive);

      env = makeEnv(c,method);
      c.setFields( c.getSavedFields() );
      c.setMethod(method,env,full-pass);
   }

   //normal path
   statement(env,method);
}
```

This code is intended to do a normal (single pass) check on any non recursive method. However, when a recursive one is given it must first remove the recursion branches and do a preliminary check before the final and complete verification (with all the recursive field states information available).

As a consequence of the internal calls presented in the previous section, this kind of recursion might occur on multiple depth levels in the internal import stack. Thus, each must take into consideration the current state of the stack to remove the recursion in relation to all the methods that are already inside it.

```
/** pseudo code for "src/yak/type/util/Context.java:356" */
checkInternalCall(MethodChecker checker,Value caller, MethodValue m){
   if( stack.contains( method ) ) {
      //stack hit -> only happens on full-pass
      Context c = stack.get( method );

      //check before call
      for( Value v : self.getFields(),c.self.getFields() ){
         if( !self.v.hasSimulation(c.self.v) )
            throw ERROR;
      }

      //before is ok, then jump to after call.
      c.getAfterCallFieldsSnapshot().copyTo(this);
   }
   else {
      //stack miss, check cache
      CacheNode n = new CacheNode(method,this);

      if( return_cache.contains(n) ){
         //if some similar call sometime before
         Context cached_c = return_cache.get(n);

         //before call compatibility was used on get, so no need to check again
         cached_c.getAfterCallFieldsSnapshot().copyTo(this);
         return;
      }

      //needs to be push new checking method
      Context new_c = this.clone();

      stack.put(method, new_c);
      checker.checkMethod( method , new_c);
```



```
class C{
    usage m

    T#a t;

    C(){
        t = new T();
    }

    m(){
        if( ? ){
            m();
        }else{
            t.a();
        }
    }
}
```

```
m(){
   ?;
   t.a();
}
```

```
m(){ // t = T#a
   ?;
   t.a();
} // t = T#stop
```

```
m(){ // t = T#a
   if( ? ){
      // t = T#a
      m();
      // t = T#stop
   }else{
      // t = T#a
      t.a();
      // t = T#stop
   }
}
```

Figure 4.22: Recursive method handling.

```
        return_cache.put(n, new_c);
        stack.remove(method);
    }
}
```

For checking an internal call it first tests if it is in the internal call stack. If so this means it is after the removed recursion state and therefore it must already know the final states of the field variables after the call has ended. As such it just checks if the current state is compatible with the calling one and jumps all local variables to the after call state. When there is a stack miss it needs to check the local call cache. This cache is indexed based not only on the method that is being called but also the state of each field variable and hence there is no need for checking the compatibility of the before call behaviors (as they are forcefully the same). A cache miss will mean that there is another method to test with a (still) unknown field state result.

Figure 4.22 shows a simple case of recursion and how it is handled by the type system, when checking the consistency of class C at the method m.

4.4 Implementation of the Interpreter

The interpreter follows a program structure similar to the type checker as both implement the visitor pattern for the more relevant AST nodes. Therefore, it basically travels through the parsed tree and evaluates each statement accordingly. Most of this evaluation process is pretty standard and as such we will refrain from presenting a detail description.

All operators are internally translated into method calls that will then do the expected calculations on the target object. Note that we don not allow for this operators to be redefined and as such this translation is mostly transparent to the programmer. This is also true for all logic operators (AND, OR). For that reason a logic operator will not stop when the remaining calls became redundant. In other words, even if an AND term is evaluated to false the interpreter will still evaluated the right term as it was internally translated to an appropriate method call

($A\&\&B = A.\$AND(B)$) without any conditional evaluation.

Every object is only accessible through its reference. This reference can either be local (pointer) or remote (location path - URL). Arguments are passed as copies of these references to any method call and because of this most operations will return a modified copy of the object instead of changing directly the target element. This is mostly visible in the XML operations as the copy must be carried on on each modification. Although it might seem unneeded, this will help the distribution abstraction. If we did not use this, the code could produce different results based on its actual location (since we do not expose basic types to the outside world and thus can not receive invocations).

As said before, there are a set of errors that can occur at run-time and will not be handled by the type checker (namely errors in the constructions of the XML objects, broken communications, null pointer exceptions) which may cause abnormal program termination.

On the protocol side, we also provide an additional testing feature to re-assure the behavioral correctness of an object's use. This means that when an object is created (at run-time) it will also carry the same automaton as the declared in the usage of the class. Therefore, each method call will be first checked against the allowed transitions. However, it does not verify if the objects behavior is completed at the time of destruction. This is intended as more of type checker debug feature than an actual language functionality.

4.5 Principles of the distribution mechanism

All non basic types are automatically registered and exposed through a web interface by the servlet. This means that upon creation any new object can be accessed from the outside. It is also possible to invoke the constructor of a class remotely. Although this can cause behavioral interferences (since we do not have any kind of permission system to differentiate among the true owner and everyone else) it also allows for some other types of clients (like web browsers) to use the system. Consequently, it should make the use of these web services more easily accessible on other scripting languages like JavaScript.

The mapping of an element to its URL format is handled transparently and it is visible at run-time when the $toUri$ method is called. It follows a simple structure, inspired by the REST (Representational State Transfer) methodology, which splits each resource into a well defined URL. The basic logic of these paths is as follows:

```
   type signature - http://URL:PORT/yak/TYPE
      method call - http://URL:PORT/yak/TYPE/INSTANCE-NUMBER/METHOD
class constructor - http://URL:PORT/yak/TYPE//CONSTRUCTOR

static signature - http://URL:PORT/yak/STATIC-VAR-NAME
     static call - http://URL:PORT/yak/STATIC-VAR-NAME/METHOD
```

Note, however, that all code is still executed only on the server side by receiving appropriate method calls to previously created objects. We also do not have any security feature for authentication or any kind of access control to the server interface. Any call to the server is redirected to a specific internal object based on the URL that was given on the request. This follows a predictable structure that obeys a similar idea to the REST methodology (although far from being REST-full).

All communication between clients and servers are handled by XML encoded messages except on HTTP-GET as the parameters must be placed on the request URL. Thus, any argument passed by a HTTP-GET must encode its values after the normal URL by appending them after an initial '?' separator and each obeying the format $argX = value$ where X is the argument number that should start at 0. If there's more than one argument it must be split with the character '&'. Example: `?arg0=12&arg1="asdasd"&arg3=true`. On a HTTP-POST the content is carried on the body of the request over a simple XML message that uses the "yak" namespace. Note that we do not check incorrect uses of the XML type that may collide with this namespace and therefore it is possible to intentionally mask these calls and mess up the whole communication by creating an XML object in that said namespace.

An example of the XML format in a HTTP-POST message:

```
//target URL: http://URL:PORT/yak/Screamer/12/repeatScream
<args xmlns="yak">
  <integer>12</integer>               //1st argument
  <string>I scream icecream!</string> //2nd argument
</args>
```

Besides the run-time communication layer, the typechecker also needs to obtain all remote type's signatures in order to check their use in the program. Static variables also have to provide their internal type. To do this information exchange we use a simple XML format that contains the appropriate contents (mostly method signatures and usage protocol). Instead of using a more standardized format like WSDL we opted for this smaller and simpler approach although future work could be done to link these two types of services.

A simple example of a class' XML description:

```
//target URL: http://URL:PORT/yak/Screamer
<class xmlns="yak" xmlns:yak="yak" yak:name="Screamer">

  <usage yak:regex="stop" />

  <constructor />

  <method yak:name="repeatScream">
    <parameter yak:type="ineteger"/>
    <parameter yak:type="string"/>
    <return yak:type="string"/>
  </method>
  <method yak:name="toString">
    <return yak:type="string" />
  </method>

  <method yak:name="toUri">
    <return yak:type="string" />
  </method>

</class>
```

The HTTP server was created by extending a Java servlet and as such this simplifies most of the server communication. Therefore, the code in `src/yak/servlet/YakServlet.java` is mostly related to forwarding the requests to the appropriate objects and answering back (there is not much of HTTP specific functions).

4.6 Remarks

In this chapter we introduced the implementation by describing the most important algorithms and techniques used in the prototype. We focused our approach to the most relevant aspects of the code in hopes that this will serve as a sufficiently detailed description for anyone interested in looking into the complete source code (available with a BSD style license in [29]).

This code shows how we handle the behavior in accordance to the different flows that may appear in a program of the developed language. Even though they are not particular lousy algorithms, these are an initial approach to the problem and therefore they all share some rough edges. This is also the consequence of taking a more general view of the problem instead of relying on specific implementation details to provide some optimizations.

Although the overall type checking complexity can take an exponential time to complete for certain AST depths, we tried to reduce each specific flow analysis to more sane levels as much as possible. This also explains why we did not chose some other (possibly easier to code) alternative solutions. For example, it would be possible to implement the **`if else`** type checking rule by just extracting all possible combinations of code flow and check each one individually. Since we also lack some more practical comparison data, it is unclear if this solution (although much worst in terms of complexity of the algorithm) has any weight in the average code checking time. This situation may occur in small programs with simple branching on some protocol sizes on which the intersection operations could actually be more time consuming than just expanding all branching cases. As a consequence of this lack of use data, the main idea of this prototype is to have a better overall solution for a more abstract case and ignore any particular optimizations of some specific situations. Therefore, more professional implementations of this type system may require some field testing in order to establish possible optimization points to improve the overall run-time.

Although we were mostly centered on the typecheck, we also briefly present the general idea of the interpreter and the distribution system. These are mainly slightly modified versions of the work develop for the diploma thesis [26] and as such this functionality is mostly inherited from it and not a specific feature specially created for this dissertation.

More detailed information can be found in the appendixes: a quick user guide with information on how to compile (C.1), run (C.2); the complete examples (appendix D); and the list of files (appendix E, which includes source files).

5

Conclusions

In this dissertation we have presented a typing system that guarantees the correct use of objects in accordance to their behavior - described by a regular-expression-like protocol.

We have shown a strict formalization of the typing rules and described how and what restrictions they force the code to obey. Our rules cover all normal sequential constructions usually found in any regular object oriented programming language. Therefore, we have proposed conditions that assure the behavioral correctness in the event of a code branch (**`if else`**), loop (**`while`**/**`repeat`**), exceptions (**`try catch`**, **`throw`**), object exchange (assign, argument, **`return`**) besides the normal object code (class consistency, etc).

As a proof-of-concept, we created a prototype that uses a simplistic language on which the behavioral typechecker can reason about the program flow correctness. We described the most important algorithms of this prototype and also expand the type system to account for some additional syntax sugar in order to make the language more user friendly.

We wrote a few small examples and also ported two from the WS-CDL specification to show how the language and type system can model and handle different behavioral situations.

Unfortunately, due too lack of time, we were not able to dive deep into the problems of type checking concurrent behavioral programs. Even though we have some syntax in the language (and interpreter) to allow for parallel composition of statements and the forking of an expression evaluation into a separate thread, we do not have appropriate typing rules for them, yet. Nonetheless, the original article [10] and others have some interesting guide lines that can be used for future work. We also have some drafts of a more particular solution to this problem that can be applied to this language, namely concurrency abstract which is briefly described in the following section.

Since we hardly mention Web Services throughout this text, it might look like the reference to this topic was somewhat unnecessary. This is not completely true as they served as an important design goal, namely in eroding the distinction between a class/service (with implications in how a class can be used/checked internally) and the communication abstraction that is present in the prototype. Although we did not get to expand on some more service related ideas, some of the most important ones are mentioned in the future work section.

From our point of view, we have fulfilled all objectives as we created a behavioral type system and a working prototype for the created language and typing rules. Even though we are far from completely solving the more broader problem of combining services, we think this work serves as a good step in that direction by proposing several solutions to more practical situations which can be used as building ground for more complex systems.

5.1 Future work

As with any work, there is always some room for improvements and some other interesting research problems that should be covered in future work, with various priorities.

Soundness proof: Although we have presented a strict formalization of the type system, there is still no guarantee these rules are sound until a complete proof is made. Therefore, one of (probably the most important) future work is to prove the soundness of this type system. We intend to use a logic framework like CELF [39]/TWELF [40] to assist this process.

Query by protocol (contract): This feature would allow for dynamically fetching an object whose protocol is consistent with the query. It would allow for a kind of "downward cast" since it returns an object with a compatible behavior contained in a generic object pool.

Although most of the infra-structure needed for this should be mostly in place (since it is just comparing types), there is no syntax or even a concrete method to use it directly.

A possible practical use could be (using a modified version of the machines example):

```
class Main{
   map<?> machines;

   main(){
      //adds lots of machines with different protocols
      machines.put(...); //(simplified)

      //dynamic query for an object with a compatible protocol
      try{
         //fetches a machine with a Bender-like protocol
         Bender worker = pool @ ((lock;(bend*+twist);release)*;standby)*;
         /*
         this would simplify the machines example, since it would not be needed
         to have different maps for each type of machine and would also allow
         for more generic machines (that can have more than one function, like
         a bender-welder machine, all-in-on machine, etc.).
         */
         //... (hidden use and return to pool)
      }catch(NotFound error){ }
   }
}
```

Parametric protocols: There is also an important missing feature in the current prototype that limits the code modularity when it is needed to store a behavioral object for a "limited" amount of time. Something along the lines of receiving a behavior, using it with a specific subset protocol, and then returning it to someone else outside the class. A possible solution to this model would be to allow a kind of parametric protocols.

Example:

```
class Machine{
   usage enter;bend;exit

   //parametric protocol, only the beginning is "known"
   Block#bend;<T> block;

   //stores block inside machine (no need to pass it as
   //an argument on the following methods like in the machines example).
   enter(owned Block#bend;<T> block){
      this.block = block;
   }

   bend(){
      block.bend(...); //(simplified)
   }

   /*
   after using the block as specified by this class' internal
   behavior, it can be returned with a remaining behavior that,
   although unknown in here, has a context given by the previous
   "enter" call. This would allow for a more generic use of
   blocks without the need to know the complete behavior when
   saving it on a class field.
   */
   Block#<T> exit(){
      return block;
   }

}
```

Behavioral dependencies: As also described in [8], there is no relation hold among interdependent behavioral objects. That is, objects whose behavior is influenced by the behavior of other(s). This is an usual design pattern for the stream readers that wrap one or more streams. Although it is possible to break these dependencies and force behavioral linearity (like in this prototype), it might be too restrictive to some uses. Thus, another possibility would be to explore the use of behavioral dependencies among different behavioral objects.

Example:

```
class Main{
   main(){
      File file = new File(...); //(simplified)
      file.openR();

      //instead of requiring ownership it just "shadows" it.
      FileReader reader = new FileReader(file);
      /*
      imagine "reader" also holds some kind of pointer to the
      behavior of "file". Therefore, when the file is closed inside
      "reader", the change is also visible on the outside since
      internally it closes "file". The reverse situation could also
      happen, when "file".close also closes "reader".
      */
      reader.close();

      file.close(); //ERROR: illegal call.
   }

}
```

Some kind of behavioral inference: This could be more of a development tool or just an exercise to add behavioral notation to legacy code. In some situations it is possible to obtain the protocol directly from a source code event without the additional notations. Although this can be complicated when there is recursion and even impossible when there is interface abstraction to account, it could have some practical uses. The search for the protocol would have to settle on finding the most restrictive version possible as otherwise there would be too many possibilities to consider.

```
m(A#? v){ //unknown protocol
   v.a();
   if( ? ){
      v.b();
   }else{
      v.c();
   }
   //v inferred initial protocol: A#a;(b+c)
}
```

Concurrency: There is no support for concurrent behavior in this type system (although there is syntax for threads/forks and concurrent composition in the interpreter - which will probably be proved insufficient to cover all concurrency possibilities and thus requiring some additional operations). This point is much larger (and harder) than any of the previous ones and it could also require a more close implementation of the article [10] namely on the concept of owned protocol paths, etc.

Concurrency abstraction: Another interesting possibility albeit with much more work and possibly with less certain results would be to use the behavioral information as a dependencies inference tool to improve the concurrency of a program. In this situation the syntax would remove all (explicit) concurrency constructions (i.e. no forks or even parallel composition) except for some new syntax in the protocol. Thus, the protocol would substitute some of the concurrency control that now is given on the statement level. The type checker would not only verify the correctness of the code according to the given protocols but also identify the concurrent possibilities that could be later used (at compile or at run-time - or both). Then something like a concurrency overlord/scheduler would decide how to best distribute the work in accordance to the currently available resources (hardware, network connections, etc). The programmer would only lose explicit concurrent control, however, it would still be possible to impose dependencies that would then have some influence to the decisions made by the overlord.

Example 1 (sequential code):

```
m(A#a;b;c v){
   v.a();
   v.b();
   v.c();
}
```

In this situation, the behavioral dependencies on all called methods are simply linear ones and thus the overlord would see that each call requires the previous one. As such, this would most likely be launched on one (single) thread.

Example 2 (concurrent calls):

```
m(A#a|b|c v){
   v.a();
   v.b();
   v.c();
}
```

Note the difference in the protocol. Now the three method calls are concurrently independent, they do not really require each other's in any way. Thus the overlord is free to launch up to three different threads (for calling each one of those methods). This would most likely also require some kind of heuristic based on the complexity of each method to better balance when it is not possible to launch the optimal number of threads.

Example 3 (controlling concurrent/sequential code):

```
m(A#a|b|c v){
   if( ? ){
      A#(a;b)|c tmp = v;
      tmp.a();
      tmp.b();
```

```
        tmp.c();
    }
    ...
}
```

Concurrency control would still be available although based on the protocol and the subtyping relation. Therefore, this would serve as a kind of dependency redefinition operator as in this example it explicitly forces the v type to change the behavioral dependencies to now require the call of the a method before b although c remains free to be interleaved.

Example 4 (expression dependencies):

```
m(A#a|b|c v){
    v.a();
    v.b( v.c() );
}
```

This example shows one of the dependencies that might appear on a program and that will influence the allowed concurrent flow. Since the call to c is an argument to the b method call (and since we do not use lazy evaluation - which would be really messy to handle) the typechecker would then infer a dependency of $a|(c;b)$ on the program.

Example 5 (concurrency split):

```
m(A#a|b v){
    m(v,v);
}

m(A#a a, A#b b){
    a.a();
    b.b();
}
```

The actually used overlord could be made hardware specific as it should adapt to the available resources. It could also transparently use the network to distribute the code or even decide which hardware best fits the calling code (like using the GPU on some streaming functions, etc). Since it would still always be possible to revert to the single threaded model (with special controls like semaphores, etc) the code could then adapt to many run environments without requiring code rewrites or recompiles.

For this concurrency model to work, each class has to be a strictly defined closed environment so that all local side effects can be controlled. It could then completely control the use of common (and remember always private) field variables, (since the concurrent context is known) with a locking mechanism based on two-phase locking (for example).

Example 6 (auto locking):

```
class C{
    usage a|b

    integer a,b,c,d;
    a(){
        a = 1;
        Lib.println( b );
        b += c;
    }

    b(){
        a = 2;
        Lib.println( b+d );
    }
}
```

In this case variable a needs write lock, b write lock on method a and read lock on b. Variables c and d have no concurrency issues. Note that for this we would always assume any interference as being unwanted (unintentional), some notations to override this would probably also have to be created.

For this to work flawlessly we would have to remove some remaining interference points of our language that may arise from the (always public) static variables. A simple work around would be to push them inside a class which will then also require for static methods.

```
class X{
    static integer s;

    static integer getS(){
        return s;
    }
}
```

Additionally, this would have some interesting possibilities for (internal) class concurrency control. We already have in place queues that can be seen as channels of communication between (possibly different) methods (inspired by the π-calculus' channels). With appropriate usage inference algorithms for these queues it is possible to detect the existence of dead-locks [41].

```
class C{
    usage a|b

    queue<integer> q;

    a(){
        q << 1;
        q >>;
    } // q#<<;>>

    b(){
        q >>;
    } // q#>>
    // dead lock on a|b, 1 push 2 pops.
}
```

For this, the queue protocol would have to be inferred from its use inside the method. This limits the case for returning or using queues as an argument without adding specific protocol notations for these elements. In other words, to allow for queue (channel) mobility we would have to add a kind of protocol notations seen in session types or similar.

All these new rules might also simplify the verification needed for concurrency since it would mostly rely on the subtyping relation. Nonetheless it still would require some extensions to the automaton which might not be that easy to accomplish or it might even need to combine with something like Petri nets. There are also other inspirational sources for this kind of handling of parallel constructions (although not quite the same) in some hardware description languages (like VHDL, etc).

Although we have presented this idea as if it were the only concurrent control available, it is probably a too extreme approach to completely remove this decision from the programmer. Therefore, for it to coexist peacefully with our normal composition syntax (';' sequential and '—' parallel) it would require an explicit notation to express that the statement composition can be handled by this automatic controller. The general idea follows the method-as-a-process concept proposed by Nierstrasz but takes it a step further by creating a similar abstraction level for concurrency control in a similar fashion to what is now common for memory management (with automatic garbage collectors, etc). Even if it would still be rather limited and simplistic, it could provide some interesting code abstraction possibilities.

Improved error messages: Although we tried to express the reason of failure in a clear and direct way it could be further improved specially with better reporting of the allowed protocol. This needs an additional algorithm to build back the behavior expression based on the current valid automaton. Even though a quadratic algorithm is already known to do this for normal automata, it is still needed to expand it in order to include our exception and recursion mechanisms. When this problem is solved, we can then remove the current temporary solution of showing the behavioral history of a type.

Other improvements: The servlet does not prohibit colliding behaviors from concurrent access to the same object from different remote sites. This is somewhat related to the previous point but could be solve by requiring an ownership identification which would allow access to the restricted methods (that is, the behavioral methods). The protocol could also be extended to include additional constructions in order to improve its expressiveness, maybe even some support for templates that expand specially limited containers to specific sizes (like creating a bag/stack of fixed dimension, etc). Finally, it would also be interesting to convert the whole type system directly to Java, maybe using Polyglot [1].

[1] http://www.cs.cornell.edu/projects/polyglot/

A

Complete Grammar

```
keywords: class interface try catch if else while repeat return static
 owned object integer double xml string boolean queue map thread fork
 throws throw usage use this null true false stop new

parseExpression ::=  Expression <EOF>
parseProgram ::=  ProgramUnit <EOF>
parseLiteral ::=  Literal <EOF>
parseRegex  ::=  getRegex <EOF>

ProgramUnit  ::=  ( ( Declarations ) | ( StaticVariableDeclaration ) )*

      IfStatement ::= if "(" Expression ")" BodyStatement ( else BodyStatement )?
   WhileStatement ::= while "(" Expression ")" BodyStatement
  RepeatStatement ::= repeat BodyStatement
  ReturnStatement ::= return ( Expression )?
   ThrowStatement ::= throw Expression
    BodyStatement ::= "{" ( StatementSequence )? "}"
TryCatchStatement ::=  try BodyStatement ( Catch )+

Catch ::= catch "(" ( ( SimpleType ) | ( ObjectType ) ) <IDENTIFIER> ")"
      BodyStatement
 Fork ::= fork Expression

Declarations  ::=  ( ClassDeclaration | InterfaceDeclaration )
    ClassDeclaration ::= class <IDENTIFIER>
            ( ( "@" StringLiteral ) | ( "{" ClassBody "}" ) )
InterfaceDeclaration ::= interface <IDENTIFIER>
            ( ( "@" StringLiteral ) | ( "{" InterfaceBody "}" ) )

ClassBody  ::=
     ( UsageDeclaration ( UseDeclaration )* )?
     (   ( VariableDeclaration )
       | ( MethodDeclaration )
       | ( QueueDeclaration )
       | ( MapDeclaration ) )*

InterfaceBody  ::=
     ( UsageDeclaration ( UseDeclaration )* )?
     ( MethodDeclaration ";" )*

   QueueDeclaration ::= queue ComplexTypeDeclaration
            <IDENTIFIER> ( "," <IDENTIFIER> )* ";"
     MapDeclaration ::= map ComplexTypeDeclaration
            <IDENTIFIER> ( "," <IDENTIFIER> )* ";"
VariableDeclaration ::= TypeAnnotation
            <IDENTIFIER> ( "," <IDENTIFIER> )* ";"

StaticVariableDeclaration ::= TypeAnnotation StaticVariable
               ( "," StaticVariable )* ";"
StaticVariable ::= <IDENTIFIER> ( "@" StringLiteral )?

SingleVariableDeclaration ::= <IDENTIFIER> ( "=" Expression )?
VariableWithInitializerDeclaration ::= TypeAnnotation SingleVariableDeclaration
               ( "," SingleVariableDeclaration )*

MethodDeclaration ::= TypeAnnotation? <IDENTIFIER> ParameterList ThrownList
            "{" ( StatementSequence )? "}"

ThrownList ::= ( throws ThrownType ( "," ThrownType )* )?
ThrownType ::= ( <IDENTIFIER> | integer | double | string | boolean | xml )

ParameterList ::= "(" ( Parameter ( "," Parameter )* )? ")"
Parameter ::= TypeAnnotation <IDENTIFIER>

Literal ::= <INTEGER_LITERAL>
  | <DOUBLE_LITERAL>
  | <BOOLEAN_LITERAL>
  | <NULL_LITERAL>
  | StringLiteral
  | XMLLiteral

StringLiteral ::= <STRING_LITERAL>

XMLLiteral ::= "<" XMLName ( XML_Attribute_Content )* ( ( "/>" ) |
             ( ">" ( XML_Content )* "</" XMLName ">" ) )
XMLName ::= XMLNameStr | <STRING_LITERAL>
XMLNameStr ::= <IDENTIFIER> ( ":" <IDENTIFIER> )?
XML_Content ::= Literal | "(" Expression ")"
XML_Attribute_Content ::= XMLName "=" XML_Content
```

```
AllocationExpression ::= new <IDENTIFIER> ArgumentList

StatementSequence  ::=  StatementComposition ( ( ";" ( StatementComposition )? ) )*
StatementComposition  ::=  Statement ( "|" Statement )*
Statement ::= IfStatement | WhileStatement | RepeatStatement
   | ReturnStatement | Fork | ThrowStatement | TryCatchStatement
   | VariableWithInitializerDeclaration | StatementExpression

Call ::= <IDENTIFIER> ArgumentList

ArgumentList ::= "(" ( Expression ( "," Expression )* )? ")"

StatementExpression ::= LeftReferenceAssignmentAndQueueOps | StatementMethodCall
StatementMethodCall ::= StatementContextMethodCall | StatementSimpleMethodCall
StatementContextMethodCall ::= ( ( this | static ) "." )?
   ( IdentifierWithTailHead "." )? Call ( StatementMethodCallSequence )?
StatementSimpleMethodCall ::= ( AllocationExpression | "(" Expression ")" | Literal )
      StatementMethodCallSequence
StatementMethodCallSequence  ::=  ( "." Call )+

Expression ::= ( Fork | LeftReferenceAssignmentAndQueueOps | ConditionalExpression )
ConditionalExpression ::= ConditionalOrExpression
   ( "?" Expression ":" ConditionalExpression )?
ConditionalOrExpression ::= ConditionalAndExpression
   ( "||" ConditionalAndExpression )*
ConditionalAndExpression ::= EqualityExpression ( "&&" EqualityExpression )*

EqualityExpression ::= RelationalExpression
      ( ( "==" RelationalExpression ) | ( "!=" RelationalExpression ) )*

RelationalExpression ::= AdditiveExpression
      ( ( "<" AdditiveExpression ) | ( ">" AdditiveExpression ) |
      ( ( "<=" | "=<" ) AdditiveExpression ) |
      ( ( ">=" | "=>" ) AdditiveExpression ) )*

AdditiveExpression ::= MultiplicativeExpression
      ( ( "+" MultiplicativeExpression ) | ( "-" MultiplicativeExpression ) )*

MultiplicativeExpression ::= UnaryExpression ( ( "*" UnaryExpression )
      | ( "/" UnaryExpression ) | ( "%" UnaryExpression ) )*
UnaryExpression ::= ("-" UnaryExpression) | ("!" UnaryExpression) | RightReference
AssignmentAndQueueOps ::= Assignment | QueueOps
Assignment ::= ( ( "=" Expression ) | ( "*=" Expression ) | ( "/=" Expression )
      | ( "%=" Expression ) | ( "+=" Expression ) | ( "-=" Expression ) )

QueueOps ::= ( ">>" ( LeftReference )? )
  | ( "->" ( LeftReference )? )
  | ( "<<" Expression )

IdentifierWithTailHead ::= ( <IDENTIFIER> ( ".." )? ) | ( ".." <IDENTIFIER> )
```

```
LeftReference ::= ( ( this | static ) "." )? LeftIdentifierWithTailHead
LeftReferenceAssignmentAndQueueOps  ::=  ( ( this | static ) "." )?
      LeftIdentifierWithTailHead AssignmentAndQueueOps
LeftIdentifierWithTailHead ::= ( <IDENTIFIER> ( ".." )? ) | ( ".." <IDENTIFIER> )

RightReference ::= ( Literal | AllocationExpression | ( "(" Expression ")" )
   | ( static "." <IDENTIFIER> ) ) ( StatementMethodCallSequence )?
   | ThisCase | SimpleCase
ThisCase ::= this ( StatementMethodCallSequence |
   ( "." IdentifierWithTailHead ( StatementMethodCallSequence )? ) )?
SimpleCase ::= ( Call | IdentifierWithTailHead ) ( StatementMethodCallSequence )?

UsageDeclaration ::= usage getRegex
UseDeclaration ::= use <IDENTIFIER> "=" getRegex

getRegex ::= RegexAnd

RegexAnd ::= RegexOr ( ";" RegexOr )*
RegexOr ::= StateLiteral ( "+" StateLiteral )*
State ::= <IDENTIFIER> | stop | ( "(" getRegex ")" ) | ( "&" <IDENTIFIER> State )
StateLiteral ::= StateSuffix ( StateException )?
StateSuffix ::= State ( ( "*" ) | ( "?" ) | ( "??" ) | ( RegexRange ) )?
StateException ::= "[" StateExceptionSingle ( "|" StateExceptionSingle )* "]"
StateExceptionSingle ::= ( <IDENTIFIER> ( "," <IDENTIFIER> )* "->" )?
   ( <IDENTIFIER> | integer | double | string | boolean | xml ) ( ":" getRegex )?
RegexRange  ::=  "{" <INTEGER_LITERAL> ( "," ( <INTEGER_LITERAL> )? )? "}"

ObjectType  ::=  <IDENTIFIER>
ObjectAnnotationType  ::=  ( owned )? <IDENTIFIER> ( ObjectRegexAnnotation )?
ObjectRegexAnnotation  ::=  ( ( "$" <IDENTIFIER> ) | ( "#" getRegex ) )

ComplexTypeDeclaration ::= ( "<" TypeAnnotation ">" )?

TypeAnnotation ::= ( ( SimpleType ) | ( ObjectAnnotationType ) )

ThreadType ::= thread ( "<" TypeAnnotation ">" )?
SimpleType ::=
  ( ( <INTEGER_TYPE> | <DOUBLE_TYPE> | <STRING_TYPE>
    | <BOOLEAN_TYPE> | <XML_TYPE> | <OBJECT_TYPE> ) )
  |  ( ThreadType )
```

B

Behavioral Protocols

stop
empty behavior.

a+b
choice (’a’ or ’b’).

a;b
sequential (’a’ then ’b’).

a?
optional (’a’ or not).

a*
repetition (’a’ zero or more times).

a{n,m}
’a’ at least n and no more than m, where n and m are positive integers.

(a;b)??
same as ’a;b’ but allowed to **stop** anywhere.

&l(r; **stop**+l)
recursion (’r’ may be followed by **stop** or unfold the label ’l’).

a[E: e]; b
exceptions (’a;b’ when no exception is thrown, ’a;e’ when
exception E is raised)

C

Quick User Guide

The program requires the Java Runtime Environment at least version 1.6 to run and the same version of the Java Development Kit to compile. We provide the set of additionally required libraries in the `libs` directory of the downloadable archive [29]. There's also some configuration files for the servlet and examples (some of them just special tests).

C.1 How to compile

To simplify the compilation process we provide an `ant` make file that should be enough to compile and (re)generate the `jar` file. Most of the compilation is just normal stuff, however there was an issue found when extending the automaton library that had to be worked around by importing the extra classes into that jar. Note that the ant file does this automatically.

The parser generation must be done by hand with the `JavaCC` utility, and there's also some minor changes needed that are described in the beginning of `yak/parser/YakGrammar.jj`. This should only be needed for anyone interested in changing the parser as we give the resulting java files together with the rest of the source code.

Finally, there are five ant target builds (compile, dev, build, jar and clean) defined in `build.xml` which can be called with `ant TARGET`. The dev target is only meant to be a simpler way to test run the code by setting some symbolic links instead of having to create the jar with the previously mentioned trick.

C.2 How to run

To run the distributable jar (in the directory context created by the extraction of the archive), just run: `java -jar yak.jar OPTIONS FILES`. Running without the two last arguments should give a simple help message.

FILES loads and runs all the given files or directories (except for hidden files or those with a '.' prefix).

OPTIONS are a set of additional arguments that can be passed to set or change the run options, namely:

-port NUMBER: launches a servlet that listens to the incoming TCP port NUMBER (example: 8180).

-untyped: runs without typechecking (for debugging purposes only).

-debug: additional error information (only for debugging the interpreter/typechecker)

The following two options will not run the interpreter or typechecker, instead they just show their option and quit.

-tree: shows the parsed tree for each file.

-automaton PROTOCOLS: shows a quick and dirty representation of all the given protocols, matching the first two if possible.

Thus, a simple example for running the files/ws-cdl1.yak file just enter the command: java -jar yak.jar files/ws-cdl1.yak while on the same directory of the extraction.

C.3 Values

In this section we will introduce the basic types of the yak language and their most important operators/methods. The presentation is just meant to briefly present their interfaces and basic constructors (when applied).

C.3.1 Object

This is the generic object from which all other values inherit. Therefore, all types share these same three method signatures. Also note that this language is case sensitive.

```
/* src/yak/value/ObjectValue.java */
class object{
   string toString();
   string toUri();

   boolean ==(object o); //equals
}
```

C.3.2 Integer

```
/* src/yak/value/IntegerValue.java */
class integer{
   //comparing integers
   boolean < (integer i); //less
   boolean <=(integer i); //less or equals
   boolean > (integer i); //larger
   boolean >=(integer i); //larger or equals
   //operations
   integer +(integer i); //sum
   integer -(integer i); //subtraction
   integer -();          //minus
   integer /(integer i); //division
   integer *(integer i); //multiplication
   integer %(integer i); //remainder

   double toDouble();
   string toString();
   string toUri();

   boolean ==(object o); //equals
}
```

```
CODE                    RESULT
1                       1
0                       0
005                     5
-2                      -2
1+1                     2
5%4                     1
1.toDouble()            1.0
1.toString()            "1"
1.toUri()               null
```

C.3.3 Double

```
/* src/yak/value/DoubleValue.java */
class double{
   //comparing doubles
   boolean < (double d); //less
   boolean <=(double d); //less or equals
   boolean > (double d); //larger
   boolean >=(double d); //larger or equals

   //operations
   double +(double d); //sum
   double -(double d); //subtraction
   double -();         //minus
   double /(double d); //division
   double *(double d); //multiplication

   integer toInteger();

   string toString();
   string toUri();

   boolean ==(object o); //equals
}
```

```
CODE                    RESULT
1.0                     1.0
2.34567                 2.34567
-002.123123             -002.123123
0.0000                  0.0
1.0e2                   100.0
1.0e-1                  0.1
2.0*3.0                 6.0
2.3444.toInteger()      2
```

C.3.4 Boolean

Given the way we handle operators (as just syntax sugar for normal method calls) there's a small caveat on the use of the AND and OR operators on booleans. Because of this we don't have the kind of optimization that's usual on other languages as we may end up calling redundant AND's or OR's (when the caller object was already made to be false or true, respectively). This is an important design decision as it can have some repercussion in behavioral terms if the normal flow could break before evaluating all arguments. Note that the order of calls is, however, the same as in Java since we must first obtain the caller object and only afterwards the value of all arguments.

```
/* src/yak/value/BooleanValue.java */
class boolean{
   //operators
   boolean ! (); //not
   boolean &&(boolean b); //and
   boolean ||(boolean b); //or

   string toString();
   string toUri();

   boolean ==(object o); //equals
}
```

```
CODE                  RESULT
true                  true
false                 false
!true                 false
false || true         true
true && false         false
true.toString()       "true"
```

C.3.5 String

```
/* src/yak/value/StringValue.java */
class string{
   //comparing strings
   boolean < (string s); //less
   boolean <=(string s); //less or equals
   boolean > (string s); //larger
   boolean >=(string s); //larger or equals

   //operators
   string +(object s); //concatenation

   //these will return null on error
   integer toInteger();
   double  toDouble();
   boolean toBoolean();
   xml     toXML();

   string toString();
   string toUri();

   boolean ==(object o); //equals
}
```

Allowed escape sequences:

```
\n - new line
\t - horizontal tab
\b - backspace
\r - carriage return
\f - formfeed
\" - double quote
\\ - backslash
```

```
CODE                  RESULT
"a\\b"                "a\b"
"a"+1                 "a1"
"1".toInteger()       1
"1".toDouble()        1.0
"<asd/>".toXML()      <asd/>
"a"<"b"               true
"a"=="a"              true
```

C.3.6 XML

```
/* src/yak/value/XMLValue.java */
class xml{
   integer length();

   integer isPosXML(integer pos);
   string  getText(integer pos);
   xml     getXML(integer pos);

   xml add(object value);
   xml remove(integer pos);

   string getPrefix();
   string getName();
   string setName(string name);

   xml setAttribute(object name, object value);
   string getAttribute(object name);
   xml removeAttribute(string name);
   xml clone();

   string toString();
   string toUri();

   boolean ==(object o); //equals
}
```

```
CODE                                    RESULT
<asd/>                                  <asd/>
<asd qwe=(12+2)/>                       <asd qwe="14"/>
<"class">("12".toInteger())</"class">   <class>"12"</class>
```

C.3.7 Default library

```
/* src/yak/value/libs/DefaultLibrary.java */
class DefaultLibrary{
   double random();

   printT(object o);
   print(object o);
   println(object o);

   string read();
   string readLine();

   sleep(integer ms);

   string pack(object... args);
   string method(object from, string packed_arguments);
   string constructor(object from, string packed_arguments);

   xml fetchXML(string url);

   xml getType(string name);

   string toString();
   string toUri();

   boolean ==(object o); //equals
}
```

C.3.8 Maps

```
/* src/yak/value/MapValue.java */
class MapValue<T>{
   T get();
   T get(string label);

   T peek();
   T peek(string label);

   put(T t);
   put(string label, T t);

   integer size();
   integer size(string label);

   string toString();
   string toUri();

   boolean ==(object o); //equals

   /* iterator (only at a time) */

   iterator();
   string key();
   T value();
   boolean hasNext();
   T next();
}
```

C.3.9 Queues

```
/* src/yak/value/QueueValue.java */
class QueueLibrary<T>{
   <<(T t); //push

   //non-blocking pop
   T ->(var<T> v);
   T ->();

   //blocking pop
   T >>(var<T> v);
   T >>();

   ..get();
   get()..;

   string toString();
   string toUri();

   boolean ==(object o); //equals
}

class Q{
   queue<integer> q;

   m(){
      q << 2;   //q = [2]
      ..q << 3; //q = [2,3]
      q.. << 1; //q = [1,2,3]

      Lib.println(q);   //1
      Lib.println(..q); //3
      Lib.println(q..); //1

      integer var;
      ..q >> var; // var=3 ; q = [1,2]
      q.. >> var; // var=1 ; q = [2]
      q >> var;   // var=2 ; q = []
      q -> var;   // var=null ; q = []
   }
}
```

C.3.10 Threads

```
/* src/yak/value/ThreadValue.java */
class ThreadLibrary<T>{
   T result() throws ForkAborted;
   halt();

   string toString();
   string toUri();

   boolean ==(object o); //equals
}


class T{
   m(){
      thread<integer> t = fork 1+1;

      try{
         Lib.println( t.result() ); //2
      }catch(ForkAborted error){ };

      t = fork eternalLoop();
      t.halt();

      try{
         t.result();
      }catch(ForkAborted error){
         Lib.println("as expected");
      };
   }
}
```

C.4 Remote Values

It's also legal to use content accessible in a remote server, this content is also typechecked statically even though changes to the types are at run-time will inevitably cause unpredictable errors. Both constructions to access remote elements use the `@` before a string with the URL to the content server. As mention in section 4.5, the communication is done by means of XML over HTTP.

The URL format can be as simple as `localhost:8180` since it will automatically expand to the full version with the protocol and servlet directory (`http://localhost:8180/yak`). Note only the HTTP protocol is supported.

Remote classes and interfaces (after being imported) can be used normally as if they were just another local type. However, there's a main difference to the instantiation of a remote class as the new object will be created at the remote site. This means that all method calls to that object are in fact remote method calls tunneled through the net to the correct location.

Example:

```
class RemoteClasse@"URL"

interface RemoteInterface@"URL"
```

Remote static variables are follow a similar logic as operations over these elements is carried on to the appropriate location.

Example:

```
integer var@"URL";
```

After that, any call to the static variable named *var* will be redirected to the variable at the site given by the URL. This includes the assignment operation as this variable now becomes shared among all those clients that may access it.

C.5 Comments

As probably already deduced from previous examples, comments follow the same syntax as in the Java Language:

```
//single line comment

/*
multi line comment
*/
```

D

(Complete) Examples

These examples are intended to show how the typechecker handles a specific program structure and not the actual implementation of the service in the example. Therefore, most of the "real" code needed is not included (there's no real communication with a remote service to obtain a quote, etc) as all our examples are mostly skeletons for more complete implementations.

Additionaly, the directory `files/` contains some other less relevant examples.

D.1 Files

`files/file_exceptions.yak`

```
interface IOException{}
interface FileNotFound{}

interface File{
    usage &start((
       ( openRead ; read* ) +
       ( openWrite; write* ) +
       ( openReadWrite; (read+write)* )
       ; close
       )[ openRead, openWrite, openReadWrite
                -> FileNotFound: stop+(changeFile;start) |
          read, write
                -> IOException: close ] )

    use open_r = &start( openRead[ openRead -> FileNotFound: changeFile;start] )

    changeFile(string name);

   openRead() throws FileNotFound;
   openWrite() throws FileNotFound;
   openReadWrite() throws FileNotFound;

    string read() throws IOException;
    write(string content) throws IOException;

   close();

   integer size();
   string name();
}

class FakeFile{
    usage &start((
       ( openRead ; read* ) +
       ( openWrite; write* ) +
       ( openReadWrite; (read+write)* )
       ; close
       )[ openRead, openWrite, openReadWrite
                -> FileNotFound: stop+(changeFile;start) |
          read, write
                -> IOException: close ] )

    changeFile(string name) { }

    openRead() throws FileNotFound { }
    openWrite() throws FileNotFound { }
    openReadWrite() throws FileNotFound { }

    string read() throws IOException { }
    write(string content) throws IOException { }

    close() { }

    //non behavioral methods
    integer size() { }
    string name() { }
}

class Main{

   loopOpenRead(File$open_r file){
      repeat{
          try{
              file.openRead();
              return null;
          }catch(FileNotFound exception){
              file.changeFile(file.name()+"0");
          };
       }
   }

   main(){
      File file = new FakeFile();

      if( Lib.random() >= 0.5 ){
          loopOpenRead(file);
      }
      else{
          try{
              file.openRead();
          }catch(FileNotFound exception){
              return null;
          };
      };


     try{
          while( false ){
              file.read();
          };

          readSomeMore(file);

      }catch(IOException exception){ };

      file.close();

   }

   readSomeMore(File#(read*)[IOException:stop] file) throws IOException {
      file.read();
   }

   //no valid use, no valid forwarding/simulation...
    readSomeMoreUseless(File#(read*)[IOException:close] f)  {
       try{
        f.read();
       }catch(IOException exception){
        f.close();
       }
    }

}
```

D.2 Machines

files/machines.yak

```
class Block{
    usage (cut+bend+weld+paint)*

   use cuttable = cut?;bend*;weld*;paint?
   use bendable = bend*;weld*;paint?
   use weldable = weld*;paint?
   use paintable = paint?

    integer size;

    Block(integer size){
        this.size = size;
    }

    Block cut(integer amount){
        size = size/amount;
        return new Block(size);
    }

    bend(integer amount) { }

    weld(owned Block#weld other){
        other.weld();
    }
    weld(){ }

    paint(string color){ }

    integer size(){
        return size;
    }

}

class BlockWharehouse{
   Block getBlock(){
      return new Block(100);
   }
}

/*
 * Machines
 */

class Welder{
    usage ((lock;adjust;weld;unlock)*;standby)*

    use weldable = lock;adjust;weld;unlock
    use storable = standby;((lock;adjust;weld;unlock)*;standby)*
    lock(Block#stop b) { }

    adjust(Block#stop b) { }

    weld(Block#weld b1, owned Block#weld b2){
        b1.weld(b2);
    }

    unlock() { }

    standby() { }

   //stream
   Block$paintable stream(owned Block$weldable block){
      return block;
   }
}

class Cutter{
   usage ((adjust;cut;release)*;standby)*

   use cutable = adjust;cut;release
   use storable = standby;((adjust;cut;release)*;standby)*

    adjust(Block#stop b) { }

    Block cut(Block#cut b, integer amount){
        return b.cut(amount);
    }

    release() { }

    standby() { }

   //stream
   Block$bendable stream(owned Block$cuttable block){
      return block;
   }
}

class Bender{
   usage ((lock;(bend*+twist);release)*;standby)*

   use bendable = lock;(bend*+twist);release
   use storable = standby;((lock;(bend*+twist);release)*;standby)*

    lock(Block#stop b) { }

    bend(Block#bend b, integer angle, integer amount){
        b.bend(amount);
    }

    twist(Block#bend b, integer angle, integer amount){
        b.bend(angle);
    }

    release() { }

    standby() { }

    integer maxBend(){
        return 30;
    }

   //stream
   Block$weldable stream(owned Block$bendable block){
      return block;
   }

}

class Painter{
   usage (( enter; ((rotateLeftGun+rotateRightGun)*;paint)*; exit)*; standby)*

   use paintable = enter; ((rotateLeftGun+rotateRightGun)*;paint)*; exit
   use storable = standby;(
          (enter; ((rotateLeftGun+rotateRightGun)*;paint)*; exit )*;
          standby)*

    Block#paint* target;

    enter(owned Block#paint* b){
        target = b;
    }

    rotateLeftGun(integer r) { }

    rotateRightGun(integer r) { }

    paint(string color, integer amount){
        target.paint(color);
    }

    Block#stop exit(){
        return target;
    }

    standby() { }

   //stream
   Block#stop stream(owned Block$paintable block){
      return block;
   }
}

/*
 * Blueprints
 */

interface Blueprint{
    object build(BlockWharehouse w, Factory m);
}

class CarBlueprint{

   weld(Welder$weldable welder, Block#weld? b1, owned Block#weld? b2) {
        if( b1.size() > b2.size() ){
            welder.lock(b1);
            welder.adjust(b2);
        }
        else{
            welder.lock(b2);
            welder.adjust(b1);
        };
        welder.weld(b1,b2);
        welder.unlock();
    }

    Block cut(Cutter#adjust;cut;release cutter, Block#cut b, integer amount) {
        cutter.adjust(b);
        Block half = cutter.cut(b,amount);
        cutter.release();
        return half;
    }

    bend(Bender#lock;(bend*+twist);release bender, Block#bend* b,
        integer angle, integer amount){

        bender.lock(b);
        if( angle > 180 ){
            bender.twist(b,angle,amount);
        }else{
            while(angle > bender.maxBend() ){
                bender.bend(b,bender.maxBend(),amount);
                angle -= bender.maxBend();
            };
            bender.bend(b,angle,amount);
        };
        bender.release();
    }
```

```
object build(BlockWharehouse w, Factory m){

    //machines
    Cutter cutter;
    Bender bender;
    Welder welder;
    Painter painter;

    try{
        cutter = m.getCutter();
        bender = m.getBender();
        welder = m.getWelder();
        painter = m.getPainter();

    }catch(EmptyMap error){
        Lib.println("error: "+error);
        //intentionally kills every machine found (lame?)
        return null;
    };

    //blocks
    Block wheels = w.getBlock();
    Block car = w.getBlock();

    //wheels
    Block wheel_fr = wheels;
    Block wheel_fl = cut(cutter,wheel_fr,50);
    Block wheel_br = cut(cutter,wheel_fr,50);
    Block wheel_bl = cut(cutter,wheel_fl,50);

    bend(bender,wheel_fr,90,100);
    bend(bender,wheel_fl,90,100);
    bend(bender,wheel_br,90,100);
    bend(bender,wheel_bl,90,100);

    //weld to car
    weld(welder,car,wheel_fr);
    weld(welder,car,wheel_fl);
    weld(welder,car,wheel_br);
    weld(welder,car,wheel_bl);

    //paint here...
    painter.enter(car);

    painter.rotateLeftGun(90);
    painter.rotateRightGun(90);
    painter.paint("black",3);

    painter.rotateLeftGun(-30);
```

```
        painter.rotateRightGun(-30);
        painter.paint("yellow",10);

        Block#stop done = painter.exit();

        m.returnCutter(cutter);
        m.returnBender(bender);
        m.returnWelder(welder);
        m.returnPainter(painter);

        return done;
    }

}

class ToyBlueprint{
   object build(BlockWharehouse w, Factory m){
        Cutter#stop cutter = m.cutter();
        Bender#stop bender = m.bender();
        Welder#stop welder = m.welder();
        Painter#stop painter = m.painter();

      return painter.stream(
            welder.stream(
               bender.stream(
                  cutter.stream( w.getBlock() )
               )
            )
         );
   }
}

class Main{
   main(){
      Factory factory = new Factory();
      BlockWharehouse wharehouse = new BlockWharehouse();

      Blueprint toy = new ToyBlueprint();
      toy.build(wharehouse,factory);

      Blueprint car = new CarBlueprint();
      car.build(wharehouse,factory);
   }
}

/*
 * Factory
 */

class Factory{
```

```
map<Welder> welders;
map<Cutter> cutters;
map<Bender> benders;
map<Painter> painters;

Factory(){
    welders.put(new Welder());
    cutters.put(new Cutter());
    benders.put(new Bender());
    painters.put(new Painter());
}

returnPainter(owned Painter$storable painter){
    painter.standby();
    painters.put(painter);
}

returnBender(owned Bender$storable bender){
    bender.standby();
    benders.put(bender);
}

returnCutter(owned Cutter$storable cutter){
    cutter.standby();
    cutters.put(cutter);
}

returnWelder(owned Welder$storable welder){
    welder.standby();
    welders.put(welder);
}


Painter getPainter() throws EmptyMap{
    return painters.get();
}

Bender getBender() throws EmptyMap{
    return benders.get();
}

Cutter getCutter() throws EmptyMap{
    return cutters.get();
}

Welder getWelder() throws EmptyMap{
    return welders.get();
}
```

```
    Painter#stop painter() {
        try{
        return painters.peek();
        }catch(EmptyMap error){
        return null;
        }
    }

    Bender#stop bender() {
        try{
        return benders.peek();
        }catch(EmptyMap error){
        return null;
        }
    }

    Cutter#stop cutter() {
        try{
        return cutters.peek();
        }catch(EmptyMap error){
        return null;
        }
    }

    Welder#stop welder() {
        try{
          return welders.peek();
        }catch(EmptyMap error){
        return null;
        }
    }

}
```

D.3 Purchase

files/purchase.yak

```
interface Order{
   usage review*;buy?

   string review();
   buy();
}

/*
 * Travel
 */

class SoldOut{}
class TravelOrder{
   usage (packageAlaska+packageArtic)[SoldOut: stop]+
         (flight;hotel); (review*; buy?)

   use done = review*;buy?

   packageAlaska() throws SoldOut { }
   packageArtic() throws SoldOut { }
   flight(owned FlightOrder$done order) { }
   hotel(owned HotelOrder$done order) { }
   string review() { }
   buy() { }
}

/*
 * Hotel
 */

class HotelOrder{
   usage bookGroup+bookPenthouse+bookRoom* ;
      breakfast? ; dinner? ; (review*; buy?)

   use done = review*;buy?

   bookGroup(integer size) { }
   bookPenthouse() { }
   bookRoom(integer number) { }
   breakfast() { }
   dinner() { }
   string review() { }
   buy() { }
}

/*
 * Flight
 */

class InvalidSeat{}
class InvalidFlight{}
class NoFlyList{}

class FlightOrder{
   usage userData[NoFlyList: stop];
      &start(
         ( flightNumber[InvalidFlight:start+stop] ;
           &seat(flightSeat[InvalidSeat:seat+start+stop]) )
            +
         &choose( (destination;origin)[InvalidPlace:choose+stop] );
         returnFlight? ;
         insurance? ;
         (review*; buy?)
      )

   use done = review*;buy?

   userData(User user) throws NoFlyList { }
   flightNumber(integer number) throws InvalidFlight { }
   flightSeat(integer row, integer seat) throws InvalidSeat { }
   destination(string where) { }
   origin(string from) { }
   returnFlight() { }
   insurance(owned InsuranceOrder$done order) { }
   string review() { }
   buy() { }
}

/*
 * Rental
 */

class InvalidLicense { }

interface RentalOrder{
   usage rent[InvalidLicense: stop]; (review*;buy?)
   use done = review*;buy?

   rent(User responsible) throws InvalidLicense;
   string review();
   buy();
}

class BikeRentalOrder{
   usage rent[InvalidLicense: stop]; (review*;buy?)

   rent(User responsible) { }
   string review() { }
   buy() { }
}

class CarRentalOrder{
   usage rent[InvalidLicense: stop]; (review*;buy?)

   rent(User responsible) throws InvalidLicense{
      if( Lib.random() <= 0.5 ){
         throw new InvalidLicense();
      }
   }
   string review() { }
   buy() { }
```

```
}

class PlaneRentalOrder{
   usage rent[InvalidLicense: stop]; (review*;buy?)

   rent(User responsible) throws InvalidLicense{
      if( Lib.random() <= 0.8 ){
         throw new InvalidLicense();
      }
   }
   string review() { }
   buy() { }
}

/*
 * Store
 */

class StoreOrder{
   usage addProduct; setDispatcher+pickup ; (review*;buy?)
   use done = review*;buy?

   addProduct(Product p) { }
   setDispatcher(owned DispatcherOrder$done order) { }
   pickup() { }
   string review() { }
   buy() { }
}

/*
 * Dispatcher
 */

class InvalidPath { }

class DispatcherOrder{
   usage &start(path[InvalidPath:start+stop]);
          (largeTruck+smallTruck+auto);emergency? ; (review*;buy?)

   use done = review*;buy?

   path(string from,string to) throws InvalidPath { }
   largeTruck() { }
   smallTruck() { }
   auto(integer weight) { }
   emergency() { }
   string review() { }
   buy() { }
}
```

```
/*
 * Insurance
 */

class UnInsurable { }

class InsuranceOrder{
    usage userData[UnInsurable: stop];
        normalCoverage+fullCoverage ; (review*;buy?)

    use done = review*;buy?

   userData(User user) throws UnInsurable { }
    normalCoverage() { }
    fullCoverage() { }
    string review() { }
    buy() { }
}

class User{
   string name;
   string password;
   string address;

   map<Order> orders; //Note: this is a partial type #review*;buy?

   User(string name, string password, string address){
      this.name = name;
      this.password = password;
      this.address = address;
   }

   string name() { return name; }
   string password() { return password; }
   string address() { return address; }

   add(owned Order order){
      orders.put(order);
   }

   add(string label, owned Order order){
      orders.put(label,order);
   }

   Order order() throws EmptyMap{
      return orders.get();
   }

   Order order(string label) throws EmptyMap{
      return orders.get(label);
   }

   buyAll(){
      while( orders.size() > 0 ){
         try{
            Order o = orders.get();
            Lib.println( o.review() );
            o.buy();
         }catch(EmptyMap notfound){
            return;
         }
      };
   }

   cancelAll(){
      while( orders.size() > 0 ){
         try{
            orders.get();
         }catch(EmptyMap notfound){
            return;
         }
      };
   }

}

class Insurance{
    InsuranceOrder order(){ return new InsuranceOrder(); }
}

class Rental{
    RentalOrder order(){ return new BikeRentalOrder(); }
}

class Hotel{
    HotelOrder order(){ return new HotelOrder(); }
}

class Airline{
    FlightOrder order(){ return new FlightOrder(); }
}

class Dispatcher{
    DispatcherOrder order(){ return new DispatcherOrder(); }
}

class TravelAgency{
    TravelOrder order(){ return new TravelOrder(); }
}
```

```
class Product{ }

class Store{

   Product getProduct(){ return new Product(); }
    StoreOrder order(){ return new StoreOrder(); }
}

class InvalidLogin { }
class DuplicatedUser { }

class ShoppingCenter{

   usage (&start(
      register[DuplicatedUser: start+stop]
               +
      login[InvalidLogin: start+stop];
      logout
      ))*

   map<Insurance> insurances;
   map<Rental> rentals;
   map<Hotel> hotels;
   map<Airline> airlines;
   map<Dispatcher> dispatcher;
   map<TravelAgency> travelagencies;
   map<Store> stores;

   map<User> users;

   ShoppingCenter(){
      insurances.put( new Insurance() );
      rentals.put( new Rental() );
      hotels.put( new Hotel() );
      airlines.put( new Airline() );
      dispatcher.put( new Dispatcher() );
      travelagencies.put( new TravelAgency() );
      stores.put( new Store() );
   }

    User register(string usr, string pwd, string a) throws DuplicatedUser {
      if( users.contains( usr+" "+pwd ) ){
         throw new DuplicatedUser();
      };

      User user = new User(usr, pwd, a);
      users.put(usr+" "+pwd, user );
      return user;
    }
    User login(string username, string password) throws InvalidLogin {
      try{
         User user = users.get(username+" "+password);
         users.put(username+" "+password,null);
         return user;
      }catch(EmptyMap notfound){
         throw new InvalidLogin();
      }
    }

    logout(User user){
      try{
         users.get(user.name()+" "+user.password()); //removes placeholder
      }catch(EmptyMap notfound) { };
      users.put(user.name()+" "+user.password(), user);
    }

   playground(ShoppingCenter s){
      try{
         User u = s.register("user","password","address");

         try{
          RentalOrder o = rentals.peek().order();
          o.rent(u);
          u.add(o);
         }catch(InvalidLicense error){}
            catch(EmptyMap error){};

         try{
          TravelOrder t = travelagencies.peek().order();
          t.packageAlaska();
          u.add(t);
         }catch(SoldOut error){}
         catch(EmptyMap error){};

         try{

         FlightOrder f = airlines.peek().order();
         f.userData(u);
         f.flightNumber(555);
         f.flightSeat(23,1);

         HotelOrder h = hotels.peek().order();
         h.bookRoom(123);
         h.breakfast();

         TravelOrder t = travelagencies.peek().order();
         t.flight(f);
         t.hotel(h);
         u.add(t);
         }
         catch(EmptyMap error){}
         catch(NoFlyList error){}
         catch(InvalidFlight error){}
         catch(InvalidSeat error){};

         s.logout(u);
      }catch(DuplicatedUser error){
         Lib.println(error);
      }
   }

}

class Main{
   main(){
      ShoppingCenter s = new ShoppingCenter();
      s.playground( s );
   }
}
```

D.4 WS-CDL 1

files/ws-cdl1.yak

```
class InvalidLogin{}
class InvalidPayment{}
class OutOfStock{}

class Service{

    usage &l( login [InvalidLogin: l] ;
           &q( query;
               (q+logout+
                &p( purchase
                        [ InvalidPayment: p+logout | OutOfStock: q+logout ]
                    ;logout )
                )
            )
          )

   login(string username, string password)
        throws InvalidLogin { }

   string query(string what) { }

   string purchase(string query, double payment)
        throws InvalidPayment, OutOfStock { }

   logout() { }
```

```
}

class UseService{

    use_service(Service service){
        login(service);

        string result = service.query("what?");

        while( Lib.random() >= 0.4 ){
            result = service.query("what?");
        };

        boolean purchase = Lib.random() >= 0.5;

        if( purchase ){
            if( Lib.random() >= 0.5 ){
                recPayment(service,result, 200.23);
            }else{
                //just one try
                try{
                    service.purchase(result,200.23);
                }
                catch(InvalidPayment ex) { }
                catch(OutOfStock ex) { }
            }

        };

        service.logout();
    }

    login(Service#&p(login[InvalidLogin:p]) service){
        repeat{
            try{
                service.login("username","password");
                return null;
            }catch(InvalidLogin ex){
                //retry forever...
            }
        };
    }

    recPayment(
        Service#&p(purchase[InvalidPayment: p|OutOfStock: stop]) service,
        string what, double value){
        try{
            service.purchase(what, value);
        }catch(InvalidPayment ex){
            //retry with more money...
            recPayment(service, what, value*1.2);

        }catch(OutOfStock ex){
            //no good.
        }
    }

}

class Main{
    main(){
        new UseService().use_service(new Service());
    }
}
```

D.5 WS-CDL 2

files/ws-cdl2.yak

```
class InvalidProduct{}
class QuoteTimeout{}
class CreditFailure{}

class Seller{
    usage &s( getQuote[InvalidProduct:s] );
        updateQuote*;
        order[QuoteTimeout: stop]

    getQuote() throws InvalidProduct { }

    updateQuote(){ }

    boolean order(Buyer buyer) throws QuoteTimeout{
        CreditAgency ca = agency();
        try{
            ca.creditRequest();
        }catch(CreditFailure fail){
            return false;
        };

        Shipper sp = shipper();

        return sp.shippingRequest(buyer);
    }

    CreditAgency agency(){
        return new CreditAgency();
    }
    Shipper shipper() {
        return new Shipper();
    }

}

class CreditAgency{
    usage creditRequest[CreditFailure]

    creditRequest() throws CreditFailure { }
}

class Shipper{
    usage shippingRequest

    boolean shippingRequest(Buyer buyer){
        return false;
    }
}

class Buyer{

    quote(Seller#&s( getQuote[InvalidProduct:s] ) seller){
        repeat{
            try{
                seller.getQuote();
                return null;
            }catch(InvalidProduct e){ }
        }
    }

    work(Seller seller){

        quote(seller);

        while( Lib.random() >= 0.5 ){
            seller.updateQuote();
        };

        try{
            seller.order(this);
        }catch(QuoteTimeout o){ }
    }

}

class Main{
    main(){
        new Buyer().work(new Seller());
        }
}
```

E

File list

```
.:
doc/        //documentation
files/      //example files
lib/        //libraries and servlet config files
src/        //source code
build.xml   //ant make file

./files: //main examples
bottle.yak
factory.yak
file_exceptions.yak
file_simple.yak
machines.yak
purchase.yak
register.yak
shopping.yak
ws-cdl1.yak
ws-cdl2.yak

./files/test:
checked             //should work
failed_interpreter  //should fail on interpreter
failed_type         //should fail on typechecker
server/             //testing server
```

E.1 Source list

```
/* modified automaton operations */
src/dk/brics/automaton:
   Builder.java            //constructing the protocol with a given AST tree
   Pair.java
   YakOperations.java      //newly defined automaton operations
   ModifiedOperations.java //some minor modifications to library

/* unit testing classes (using tests in files/test directory) */
src/test:
   TestUtils.java
   YakParserTest.java
   YakAutomatonTester.java
   YakInterpreterTest.java
   YakTypeCheckerTest.java

/* main class */
src/yak/Main.java //options and servlet launching

/* AST tree */
src/yak/ast:
   ID.java         //node identification
   AST.java
   ASTFactory.java //node creator
   Visitor.java    //visitor pattern for most relevant nodes

src/yak/ast/display:
   ASTViewer.java //simple JTree-based AST visualizer

src/yak/ast/nodes:
   ASTNode.java //generic node
   ASTAssign.java
   ASTCall.java
   ASTCatch.java
   ASTConditional.java
   ASTBoolean.java
   ASTDouble.java
   ASTInteger.java
   ASTString.java
   ASTXml.java
   ASTXmlAttribute.java
   ASTFork.java
   ASTMethod.java
   ASTEqual.java
   ASTField.java
   ASTGet.java
   ASTIfElse.java
   ASTLeftReference.java
   ASTNewInstance.java
   ASTParallel.java
   ASTParameter.java
   ASTPut.java
   ASTReference.java
   ASTReturn.java
   ASTSequence.java
   ASTThrow.java
   ASTTryCatch.java
   ASTType.java
   ASTVariable.java
   ASTWhile.java

/* automaton (protocol) */
src/yak/automaton:
   YakAutomaton.java
   YakAutomatonFactory.java
   AutomatonFactory.java

src/yak/automaton/display:
   AutomatonViewer.java

/* interpreter */
src/yak/interpreter:
   YakInterpreter.java
   Interpreter.java
   InterpreterException.java

/* source loader */
src/yak/loader:
   Initializer.java      //internal types initializer
   Loader.java
   LoaderException.java
   ValueFactory.java

/* parser */
src/yak/parser:
   YakGrammar.jj
   Parser.java
   ParserFactory.java

src/yak/parser/autogen:
   ParseException.java //changed: 'extends Exception' to 'extends RuntimeException'
   SimpleCharStream.java
   Token.java
   TokenMgrError.java
   YakParserConstants.java
   YakParser.java
   YakParserTokenManager.java

/* servlet extension */
```

```
src/yak/servlet:
   AttachedInterpreter.java //additional interpreter operations
   YakServlet.java          //java servlet extension

src/yak/servlet/receiver: //communication abstraction (request receive)
   Receiver.java
   AbstractValueReceiver.java
   HttpGetReceiver.java
   HttpPostReceiver.java

/* type checker */
src/yak/type:
   YakTypeChecker.java
   TypeChecker.java
   TypeCheckException.java

src/yak/type/factory:
   ContainerFactory.java
   ObjectFactory.java

src/yak/type/util:
   CacheNode.java
   ConsistencyChecker.java
   Context.java
   ExceptionMap.java
   RecursionRemover.java
   Snapshot.java
   StateArray.java
   SubTyper.java
   TypeEnvironment.java

/* utility classes */
src/yak/util:
   BlockingLinkedList.java
   Converter.java
   Environment.java
   FatalException.java
   ListTree.java
   Log.java
   Message.java //error messages
   Slave.java
   StackContext.java
   UniqueID.java

src/yak/util/reference:
   ReferenceBuilder.java
   Register.java

/* values */
src/yak/value:
   Value.java
   AbstractValue.java
   BooleanValue.java
   DoubleValue.java
   StringValue.java
   IntegerValue.java
   XMLValue.java
   InterfaceValue.java
   ClassValue.java
   IdValue.java         //uniquely comparable value
   ExceptionValue.java
   ThreadValue.java
   ObjectValue.java
   QueueValue.java
   MapValue.java
   ReferenceValue.java
   VariableValue.java

src/yak/value/invoker: //communication abstraction (invoke)
   Channel.java
   AbstractHttpChannel.java
   HttpGetChannel.java
   HttpPostChannel.java
   RemoteInvoker.java

src/yak/value/libs:
   DefaultLibrary.java

src/yak/value/method:
   AbstractMethodValue.java
   ASTMethodValue.java
   JavaCheckedMethodValue.java //special purpose method checker
   JavaMethodValue.java        //default method checker
   MethodMap.java
   MethodValue.java
   YakVisible.java //visible method annotation

src/yak/value/packer: //value packaging
   ValuePacker.java
   PackableValue.java
   SourceValuePacker.java //as source code
   XMLValuePacker.java    //as xml
```

Bibliography

[1] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, Berlin, 1991.

[2] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.

[3] Oscar Nierstrasz. Regular types for active objects. In *OOPSLA*, pages 1–15, 1993.

[4] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[5] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In Stein Gjessing and Kristen Nygaard, editors, *ECOOP*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77. Springer, 1988.

[6] Roel van der Goot and Arie de Bruin. Syntax and semantics of procol. In Jirí Wiedermann and Petr Hájek, editors, *MFCS*, volume 969 of *Lecture Notes in Computer Science*, pages 509–518. Springer, 1995.

[7] Sebastian Pavel, Jacques Noyé, Pascal Poizat, and Jean-Claude Royer. A java implementation of a component model with explicit symbolic protocols. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Software Composition*, volume 3628 of *Lecture Notes in Computer Science*, pages 115–124. Springer, 2005.

[8] R. DeLine and M. Fahndrich. The fugue protocol checker: Is your software baroque, 2003.

[9] Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

[10] Luís Caires. Spatial-behavioral types, distributed services, and resources. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *TGC*, volume 4661 of *Lecture Notes in Computer Science*, pages 98–115. Springer, 2006.

[11] Antonio Vallecillo, Vasco Thudichum Vasconcelos, and António Ravara. Typing the behavior of software components using session types. *Fundam. Inform.*, 73(4):583–598, 2006.

[12] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *POPL*, pages 331–342, 2002.

[13] Futoshi Iwama, Atsushi Igarashi, and Naoki Kobayashi. Resource usage analysis for a functional language with exceptions. In John Hatcliff and Frank Tip, editors, *PEPM*, pages 38–47. ACM, 2006.

[14] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng.*, 28(11):1056–1076, 2002.

[15] B. Rumpe and C. Klein. Automata describing object behavior. In *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 265–286. Kluwer Academic Publishers, 1996.

[16] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In George C. Necula and Philip Wadler, editors, *POPL*, pages 261–272. ACM, 2008.

[17] Cosimo Laneve and Luca Padovani. The must preorder revisited. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 212–225. Springer, 2007.

[18] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.

[19] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. 2008.

[20] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008.

[21] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.

[22] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[23] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In Yolande Berbers and Willy Zwaenepoel, editors, *EuroSys*, pages 177–190. ACM, 2006.

[24] Simon Gay, Vasco T. Vasconcelos, and António Ravara. Dynamic interfaces. 2007.

[25] Edward A. Lee and Yuhong Xiong. A behavioral type system and its application in ptolemy ii. *Formal Asp. Comput.*, 16(3):210–237, 2004.

[26] Filipe Militão. Interpretador e sistema de tipos para uma linguagem de programação para web services (yak), 2007.

[27] Web services description language. `http://www.w3.org/TR/wsdl`.

[28] Web services choreography description language. `http://www.w3.org/TR/ws-cdl-10/`.

[29] yak home page. `http://ctp.di.fct.unl.pt/yak/`.

[30] Java home page. `http://java.sun.com/`.

[31] Jetty webserver. `http://jetty.mortbay.com/`.

[32] Java compiler compiler. `https://javacc.dev.java.net/`.

[33] Eclipse ide. `http://www.eclipse.org/`.

[34] Javacc eclipse plugin. `http://sourceforge.net/projects/eclipse-javacc`.

[35] Jdom home page. `http://www.jdom.org/`.

[36] Junit testing framework. `http://junit.org/index.htm`.

[37] dk.brics.automaton home page. `http://www.brics.dk/automaton/`.

[38] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[39] Anders Schack-Nielsen and Carsten Schürmann. Short talk: Celf – a logical framework for deductive and concurrent systems. presented at LICS, 2008.

[40] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer, 1999.

[41] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2000.