Georg Göri, BSc

Erlang-based Execution and Error Handling for Abstract Behavioural Specifications

MASTER'S THESIS

Graz University of Technology

Institute for Software Technology

Supervisor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig
Co-Supervisor: Professor Dr. Einar Broch Johnsen, University of Oslo

Graz, May 2015

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, am _____ _____

Datum Unterschrift

Abstract

Software Modeling allows to verify a system's correctness and compliance to non-functional requirements in a highly abstracted setting. Writing correct and reliable software for distributed systems is very hard, especially when they have to operate under non-optimal conditions. Some faults are unavoidable in a distributed setting, like node failures or partial network dysfunctions. Therefore, we argue that it is essential for distributed models to include error handling. In this work we add error handling to the Abstract Behavioural Specification (ABS) modeling language. ABS supports modeling distributed systems with *active objects*, which interact via asynchronous invocations. The invocation returns a *future*, which resembles a transferable container, which will contain the result after the call's completion. The error handling is inspired by successful concepts of the Erlang programming language. We therefore support Erlang's principles of error-propagation, fail fast systems and restarts in case of an error via error-containing futures, object termination and implicit rollbacks. These primitives allow us to implement Erlang-style linking between objects and on top of that supervision trees in ABS. Supervision trees are a concept, where objects are automatically recreated if they terminate. These changes are implemented on a newly written Erlang *backend*, which is a part of the ABS compiler infrastructure and generates an executable Erlang program from an ABS model. We represent the key entities in ABS with Erlang's lightweight processes and asynchronous message passing. Due to conceptional similarities in Erlang's *Actors* and ABS's active objects and a large overlap in functional parts of both languages, the backend's translation can be relatively direct and therefore has successfully been the basis of other's work. In certain models it has shown a performance gain in the order of one magnitude in comparison to the other default backends. The translation and the error handling were evaluated in a case study, that functioned as running example in this work. Results show that in an unchanged model of the case study the error handling covers most of the discussed errors and with the addition of object linking in the model all of them can be handled. We also evaluated the error handling in a distributed modeling approach, that was introduced here and implemented on top of Erlang's built-in distribution capabilities. Network errors for asynchronous calls can be reliably handled, whereas distributed futures still pose a problem.

Kurzfassung

Mittels Software Modellierung kann die Korrektheit und die Erfüllung von nicht-funktionalen Anforderungen in einer abstrakten Art und Weise überprüft werden. Das Erstellen von korrekter und zuverlässiger Software für verteilte Systeme ist im Allgemeinen schwierig, insbesondere wenn diese nicht unter optimalen Bedingungen operieren. Manche dieser Bedingungen sind nicht kontrollierbar bzw. unvermeidbar, wie z.B. ausfallende Knoten oder partielle Netzwerkverbindungsverluste. Daher ist die Fehlerbehandlung in verteilten Systemen essentiell und sollte auch in deren Modellen berücksichtigt werden. In dieser Arbeit wird die Abstract Behavioural Specification (ABS) Sprache um eine Fehlerbehandlung erweitert. ABS ermöglicht die Modellierung von verteilten Systemen mittels *aktiver Objekte* und asynchronen Aufrufen zwischen diesen. Zum Zeitpunkt des Aufrufs wird ein *Future* an den Aufrufer zurückgegeben. Dieser transferierbare Platzhalter wird am Ende des Aufrufs mit dem Ergebnis befüllt. Erfolgreiche Konzepte der Programmiersprache Erlang inspirierten die Fehlerbehandlung. Daher werden Erlang Konzepte wie Fehlerpropagierung, Terminierung im Fehlerfall und automatischer Neustart, durch Fehlerspeicherung im Future, Objektterminierung und implizite Wiederherstellung des Objektzustands unterstützt. Diese grundlegenden Operationen ermöglichen die Implementierung von *Linking* zwischen Objekten und darauf aufbauend *Supervision Trees* in ABS. Supervision Trees ermöglichen den automatisierten Neustart von Objekten im Fehlerfall. Diese Änderungen sind im neu erstellten Erlang *Backend* implementiert, welches Teil des ABS-Compilers ist und ein ausführbares Erlang Programm aus einem ABS Modell generieren kann. Die grundlegenden ABS Komponenten werden mit Erlangs leichtgewichtigen Prozessen und asynchronem Nachrichtenaustausch realisiert. Aufgrund der konzeptionellen Ähnlichkeit von ABSes aktiven Objekten und Erlangs Aktoren und der funktionalen Sprachelemente, ist die Übersetzung relativ direkt und konnte daher schon als Basis für weiterführende Arbeiten herangezogen werden. In manchen Modellen können auch Geschwindigkeitssteigerungen um den Faktor 10 erreicht werden. Die Übersetzung und die Evaluierung des Fehlermodells wird in der vorliegenden Arbeit anhand einer Fallstudie gezeigt. Die Resultate zeigen, dass in einem ungeänderten Modell die meisten, und mittels Linking alle, Fehlerfälle abgedeckt werden können. Weiters wurde die Fehlerbehandlung in einem verteilten Modell überprüft. Dieses Modellierungskonzept wurde hier eingeführt und mittels Erlangs eingebauter Verteilungsmechanismen implementiert. Netzwerkfehler können im Allgemeinen gut behandelt werden, wohingegen Fehler im Zugriff auf verteilte Futures noch unzureichend behandelt werden können.

Acknowledgments

I want to thank all people, who supported me in the creation of this thesis and throughout my studies.

My first and foremost thanks goes to my supervisors, Professor Bernhard Aichernig in Graz and Professor Einar Broch Johnsen in Oslo. Both guided me through this endeavor. Prof. Aichernig supported me in the creation of the thesis and the formulation of ideas. Prof. Johnsen introduced me to ABS and gave important input for the error handling.

I would like to express my sincere gratitude to Dr. Rudolf Schlatte and Dr. Volker Stolz. They supported me throughout my exchange semester at the University of Oslo, helped me understand the details of ABS and its implementation, and encouraged my first steps into scientific writing and presentation.

In addition, I would like to thank the whole Precise Modeling and Analysis Research Group at the University of Oslo. They welcomed me cordially, gave me advice and in general made my stay very pleasant.

Last, but not least, I want to thank my girlfriend, my friends and my family, especially my parents. All of them, who did not only support me while studying and writing this thesis, but support me on the journey called life. Thank you very much for that.

Georg Göri
Graz, May 2015

Contents

Acronyms

ABS Abstract Behavioural Specification

AST Abstract Syntax Tree

COG Concurrent Object Group

FIFO First In First Out

HATS Highly Adaptable and Trustworthy Software using Formal Models

IEEE Institute of Electrical and Electronics Engineers

JVM Java Virtual Machine

LOC Lines of Code

OTP Open Telecom Platform

PID Process Identifier

SLA Service Level Agreement

SDL Specification and Description Language

VTS Video Transcode Server

Chapter 1.

Introduction

1.1. Motivation

Software modeling can give confidence in a system's correctness and conformance to non-functional requirements, like response time or throughput, by resembling a more abstract version of a target system. A model's level of detail has to be finely balanced, to be concise and abstract enough to still show the desired properties.

As modern applications have become larger and more globally accessible, they are run in a distributed and/or virtualized setting to handle the higher and more volatile load. Due to the dynamic nature of these runtime conditions, these applications have to be more aware of their environment and be able to react on changes.

Modeling can help to test application designs regarding their throughput under certain resource limitations [JST12], but this also requires that the model is aware of its environment. For distributed systems, representing the distribution itself should be considered in the modeling phase.

Part of the environmental awareness is that nodes in distributed environments can be temporarily or permanently unreachable. The *analysis of production failures in distributed data-intensive system*, like databases and map reduce frameworks by Yuan et al. [YLZ+14] showed based on bug reports that in 24% of the examined failures an unreachable node was one of the causes. This is notably as it is a condition, which is encountered at a high rate in a large distributed system according to Gray [Gra86] and therefore a failure-free system needs to be fault tolerant. Furthermore, Yuan et al. state that 63% of the failures only occur on a system with at least two nodes, which strengthens the argument that one has to consider modeling distribution more thoroughly.

As a certain kind of theses errors are unavoidable in physical systems, they have to include routines to handle those. A study by Guo et al. [GMY+13] on outages by major cloud applications like Microsoft's Azure, Amazon's EC2, Facebook and Google's Gmail, showed how a faulty error recovery leads to system failures. This highlights that error handling is an integral part of a distributed application.

When one wants to model and analyze real-world usage scenarios regarding their resource consumption and error handling, potentially more interactions and execution steps need to be included, which results in larger models and longer execution time.

1.2. Problem Statement

The Abstract Behavioural Specification (ABS)[1] language is a statically typed object-oriented modeling language and toolsuite for distributed systems [JHS+10]. It facilitates the concept of *concurrent objects* [JOA03], where an object is concurrent in the sense that it can handle multiple method invocations with cooperative scheduling between those. For architectural reasons a set of objects can be grouped into a Concurrent Object Group (COG). Between COGs calls can only be placed asynchronously. A *future* [dBCJ07], a shareable single-assignment container for the results, decouples the callee from the caller and allows to retrieve the computed value. Furthermore, ABS's objects can be *active* by providing a special method representing the object's inherent behavior, which is activated upon its instantiation. A wide range of analysis tools for ABS exist. They support test generation, property based testing, resource analysis and formal analysis based on invariants [ABG+12].

ABS's design allows to model distributed systems by *concurrent objects*, but lacks a notation of errors and handling of them. As discussed above it is crucial to consider errors in a more detailed distributed model, as they have to be part of an implementation and therefore need to be verified as well.

Moreover, if a system's model should be closer to the implemented system, or even be the basis for it (by generating implementation level code from a model), error handling is also a worthwhile addition to a non-distributed model. An implemented system should contain error handling, because in such an environment runtime errors like *division by zero* or *out of memory* errors or even software faults in underlying components can happen. Therefore it can make sense to have a less abstract model including error handling.

Due to the reasons stated above, we want to design and implement an error model and error handling primitives for ABS, which enables in further steps to model a distributed system, where errors are well defined and can be handled. To simulate models with the introduced error handling primitives, the ABS toolsuite has to be extended with support for error handling.

[1] Even though this work is written in American English, for ABS we use the British English spelling, chosen by the HATS project

1.3. Methodology

Concurrent objects are comparable to *actors*, a model for concurrent computation [AZ98]. An actor represents a unit of execution, that can exchange messages with other actors and create new actors. A concurrent object can therefore be represented by an actor with fields as local state and asynchronous activations via messages.

Erlang is a widely used implementation language resembling the actor concept. It is especially known for its error handling capabilities and being the basis of high availability distributed software [Vin07, Arm03]. Therefore we want to look at the successful error model of Erlang and adapt it for ABS with its active concurrent object groups.

On top of the new error handling constructs in ABS, we want to be able to use supervision, a key concept for the failure-tolerance in Erlang. Supervision allows to structure the actors in Erlang and restart parts of them in case of an error.

The ABS toolsuite provides a framework, that parses and generates a type-checked intermediate representation of a model. It can be used for analysis or be translated by a so-called backend to an executable representation. Part of the ABS toolsuite are backends for Java and Maude. As the Erlang runtime comes with built-in distribution that also respect the error handling primitives, we design an Erlang backend as a means to evaluate ABS with the new error model in a distributed setup.

Motivated by the similarities of the actor and concurrent object models, we hope to gain with the Erlang backend for ABS a translation that can be simpler and more native to the target language. This should in turn result in clearer output code, that can be easily extended with more features in the future. Furthermore, as for larger distributed models execution runtime can be a limiting factor. We want to achieve a faster execution by using a simpler Erlang representation, which preserves the same level of concurrency as present in ABS and executes on the Erlang runtime, which is known for its good concurrent performance.

The path to an error model and further to an evaluation of it in a distributed manner, will be taken by firstly providing a translation to Erlang, then creating an error model with an Erlang implementation and lastly showing an approach to distribution with errors. We will do so alongside using and extending a case study. For a better understanding of especially the concurrent capabilities of ABS, the case study will be first described as a model. After introducing the translation and error model, the default error handling will be examined and new constructs plus supervision will be added.

1.4. Structure and Notation

The rest of this thesis is structured in the following way:

Chapter 2 introduces the necessary background knowledge. The ABS and Erlang languages are described and key concepts are explained.

Chapter 3 outlines the used case study and its requirements. Furthermore, shows how to model it in ABS, which highlights the simplicity of the concurrency model.

Chapter 4 covers the mapping of ABS to Erlang. First, the high level translation aspects and then a more detailed description of the Erlang representation of a model are presented. We take a close look at the runtime functions and processes, which support the execution of the Erlang translation.

Chapter 5 shows the Erlang backend, which translates a model to Erlang. How the new backend is implemented and in which way it integrates with the existing ABS toolsuite will be covered in this chapter. Some translation areas are highlighted and then a ABS class of the case study and its translation are presented. With the implementation in place the unit testing of the new translation and a guide for executing an ABS model on the Erlang backend will be given. The chapter closes with a comparison of the new backend with the standard backends of the ABS toolsuite.

Chapter 6 discusses errors, faults and failures. After outlining key concepts in Erlang, the ABS error handling and its implementation will be shown. An evaluation of the presented idea happens by introducing it to the case study and by building supervision in ABS on top of it. The latter will then be used to expand the case study.

Chapter 7 focuses on distribution. A concept how to represent distributed execution in ABS and an implementation in the Erlang backend will be discussed. Then the behavior of the new error model in a distributed error-facing execution will be evaluated and open obstacles discussed.

Chapter 8 looks at related work on error handling in concurrent systems and distributed models.

Chapter 9 summarizes the thesis and discusses the findings. Finally, an outlook on future and ongoing work will be made.

Notation Throughout this work code and examples in multiple languages are referenced and explained. To improve readability and understanding especially when referring to multiple languages in the same paragraph, different styles are used for the main languages, ABS, Erlang and Java, in this work. Examples of the used styles are shown below.

All three languages use different distinguishable fonts. Additionally different colors are used to improve readability. ABS snippets are presented on a green background and the keywords are highlighted in black. For Erlang code a white background is used instead. The background for Java source code is colored gray and keywords are printed in light blue. These styles are also used embedded in descriptive text and graphical illustrations following the same style like `ABS code`, `Erlang code` and `Java code`.

```
1 Object new_object = new Object()
```

ABS code notation

```
1 Pid = spawn( fun() -> object end ).
```

Erlang code notation

```
1 Object new_object = new Object()
```

Java code notation

Context of this work The Erlang backend and the ABS additions were developed as part of an exchange semester from August 2013 to February 2014 at the Precise Modeling and Analysis Research Group at the University of Oslo. There, Professor Einar Broch Johnsen supervised the conception and implementation. The thesis itself was mostly written in Graz under supervision of Professor Bernhard Aichernig.

Chapter 2.

Background

In this chapter we introduce the principle technologies the following work is based on. The Erlang programming language and ABS modeling language will be examined. For both we highlight key concepts that are usually not found in other common languages.

2.1. Erlang

Erlang is a dynamically typed concurrent functional programming language [Arm10]. The functional paradigm is represented by supporting single assignments, pattern matching, first and higher order functions. To enable concurrent programs, Erlang provides lightweight processes as means of evaluating a function. The processes are able to communicate by using asynchronous messages [AS88].

Erlang is shipped tightly coupled with the standard library Open Telecom Platform (OTP), which provides, besides a lot of other functionality, the necessary runtime components to execute a program distributed over multiple hosts.

These distribution facilities and the error handling capabilities, which will be explained and used in this work in Chapter 5 and 6, allow to build highly reliable and scalable services like Ericsson's AXD 301 telecommunication switch [Arm03] or WhatsApp [Wha12, Vin07].

Erlang's development was started in 1987 at the Laboratory at Ericsson Telecom AB by Joe Armstrong [Arm07a]. The goal of this research project was to develop a programming language, which better suited the application in telecommunication switching systems than existing ones did. It was used successfully internally at Ericsson, but after a strategy shift the runtime and language was released under an open-source license in 1998. Following that, a larger audience became aware of Erlang and its advantages.

The Actor Model forms the theoretical basis for Erlang processes and their communication. The concepts introduced by Hewitt et al. in [HBS73] can be used to model a concurrent system. Actors are the principle unit of computation in this model. They are independent of other actors and communicate via asynchronous messages. When an actor receives a message from another actor, it performs a local computation, which is independent of the other actors. The computation is a concurrent composition of the following base operations:

- Send messages to other actors
- Create new actors
- Specify the behavior on how to handle the next arriving message

Processes are the mechanism that allows concurrent execution in Erlang. A process is started by the `spawn(<Function>)` function, which takes either a function name or an anonymous function (often also referred to as lambda expression) as parameter. A Process Identifier (PID) referencing to the newly created process is returned at creation. The PID can later be used to send messages to this process (see below).

As the name hints, processes do not share memory amongst each other. Therefore a process can be seen as function evaluation with its own stack, heap and mailbox. This simple design does not require any costly interaction with the operating system, which allows to implement processes solely in Erlang's virtual machine with a very small memory footprint of a few hundred bytes.

Modules are Erlang's concept for structuring and grouping code. A module resides in a file named `modulename.m` and contains attributes and function declarations. These attributes allow to specify records (a named tuple data type) and exports and imports, which control the visibility of functions from and to other modules.

The code for one kind of processes is normally placed in a single module. Only functions for startup and interaction with these processes are exported from this module. Thus one can hide internals in the messaging systems by using this kind of modules. In documentation and explanations, such a process is often referred to by its module name, as it represents this module's behavior.

Message passing and its guarantees A process `A` can asynchronously send a message containing an arbitrary term `T` to another process with the PID `B`, by using the send `!` operator in the following way: `B!T` . This message will then be put in `B`'s mailbox, which is a queue of dynamic length. Messages in the mailbox can be fetched by a `receive` statement, which allows to specify multiple patterns, where a message has to match one of those patterns to be retrieved from the mailbox. If no timeout is specified the `receive` statement will block until a matching message is received.

The Erlang runtime system does neither guarantee a global order of messages in the mailbox nor that messages are reliably sent. It ensures however that messages, sent from one process to another, will arrive there in the same order. This aspect is highly important, when implementing correct protocols. Formal semantics for Erlang and its distributed execution were also proposed by Svensson and Fredlund in [SF07, Fre01].

An Example is used to illustrate the Erlang concepts presented in the previous paragraph. In the code, shown in Figure 2.1, two processes exchange messages. The example consists of multiple expressions. A single expression ends with a dot (`.`), as can be seen for the expression over the Lines 1–5. They can be evaluated by pasting them in the interactive Erlang shell (a binary called `erl`) and will be discussed expression by expression in the following paragraph.

```
1  Responder_fun=fun () ->
2                  receive {Other,ping} ->
3                          Other!pong
4                  end
5                end.
6  Responder=spawn(Responder_fun).
7  Responder!{self(),ping}.
8  receive pong ->
9          io:format("success~n")
10 end.
```

Figure 2.1.: A ping-pong example in Erlang

The first statement over Line 1–5 creates a function. It contains a **`receive`** statement, that waits for a message, which is a two-element-sized tuple. The **`receive`** performs the same kind of pattern matching a **`case`** expression does. One can specify different patterns with bound and unbound variables. The received message has to match such a pattern with its bounded variables. In case of a match, unbounded variables will be bound to the specific value of the expression. In our example the first element is bound to the variable `Other`and the second element has to be the atom `ping`. Should such a message be received, Line 3 is executed, where the atom `pong` is send to the PID bound in `Other`. A new process, evaluating this function is then created in Line 6. The **`spawn`** function returns the PID of this newly created process, which is stored in the variable `Responder`.

In Line 7 a message is send to the `Responder`. It matches what is expected by the function in Line 2. The first element of the sent tuple is the PID of the current process, which is retrieved by a call to **`self`**. As discussed above, the `Responder` replies to such a message with a message of the atom `pong`. After that the initial process waits in the receive statement over Lines 8–10. When it receives a message matching `pong`, it prints `success` to the standard output. Note that after the `Responder` has evaluated the **`receive`**, the function end is reached and therefore the server is terminated.

Behaviors are important design patterns in Erlang. The concept consists of a generic implementation for a structure that is often needed, like a long-running process that replies to messages. These implementations come along with a rich set of API functions to use them. The business logic of those is then provided by a so-called behavior module, which has a well defined set of callback functions.

The long-running process above is provided by the `gen_server` module. It handles internally asynchronous and synchronous messages with a return value to the caller, and additionally provides functions for hot-code upgrade and handling timeouts. Moreover, it maintains a term representing its internal state. The behavior module has in principle only a function to initialize this state, and a function that is called, when a message is received. This function is supplied with the internal state and has to return a term representing the updated internal state and an optional reply to the sender.

Other Erlang/OTP behaviors used in this thesis are `gen_fsm` and `gen_event`. The first is like a `gen_server` with an additional internal finite state machine, where the callbacks depend on the internal state. The `gen_event` process resembles the observer pattern. One can add multiple `gen_event` behavior modules, which get called when an event is fired.

A `gen_server` example shows how this behaviors can be used. In Figure 2.2 we display a `gen_server`-based version of the ping-pong of Figure 2.1 on the previous page. In addition to the previous example, in this one the `Responder` maintains a list of processes it has replied to. The script, starting and using the server, is shown in Figure 2.2(a). The behavior module used for this server is shown in Figure 2.2(b). First the script and then the behavior module will be explained.

In the first line the `gen_server` process is created via the `start` function. Here we have to pass the name of the module, containing the behavior callbacks, and we could pass additional arguments and options. If the startup was successful, the `start` function returns a reference to the new process as a tagged tuple. The `call` in Line 2 sends a message to the server and waits for a reply. An asynchronous alternative to this is the `gen_server:cast` function. The `call` function is invoked with the reference to the `ping_server` and a message, which is in our case the atom `ping`. The returned value of the call is the answer of the `gen_server`, which we expect in Line 7 to match the atom `pong`. After that `success` is printed to the standard output, to conform to the behavior of the previous ping-pong example.

The `ping_server` module has to provide a set of callback functions, as defined by the `gen_server` module. These functions are called for a specific situation or event and have a set of allowed return values. Via the compiler directive in Figure 2.2(b) Line 2, the compiler checks if all required functions are provided by this module. These methods also have to be exported to be available, which happens in Line 3. We will now discuss each of these functions.

```
1 {ok,Responder}=gen_server:start(
      ping_server, [], []).
2 pong=gen_server:call(Responder,
      ping).
3 io:format("success~n").
```

(a) Startup and call of the `ping_server`

```
1  -module(ping_server).
2  -behavior(gen_server).
3  -export([init/1, handle_call/3,
       handle_cast/2, handle_info/2,
       terminate/2, code_change/3]).
4
5  init([]) ->
6      {ok, []}.
7
8  handle_call(ping, {From,_}, State) ->
9      {reply, pong, [From|State]}.
10
11 handle_cast(_Msg, State) ->
12     {noreply, State}.
13
14 handle_info(_Info, State) ->
15     {noreply, State}.
16
17 terminate(_Reason, _State) ->
18     ok.
19
20 code_change(_OldVsn, State, _) ->
21     {ok, State}.
```

(b) The `ping_server` behavior module

Figure 2.2.: A `gen_server` implementation of the ping responder

The `init` function is called after the startup and has to return the initial state of this server. As we want to maintain a list of clients, the server has replied to, an empty list (`[]`) is set as the initial state. The `handle_call` function is invoked, when a client executes `gen_server:call`. The message of the client is the first parameter and the sender the second. As third parameter the term representing the server's internal state is passed. In this example we match only messages of the form `ping`. In Line 9 the return value `{reply,pong,[From|State]}`, indicates that `pong` should be returned to the caller and the server's new state is the list `[From|State]`, where the caller's PID is appended in the front of the previous list. The `handle_cast` and `handle_info` functions work similarly. The former is called, when `gen_server:cast` is invoked and the latter is invoked for every message the `gen_server` process receives, which is not handled by any of the other two functions.

When the `gen_server` process terminates, the `terminate` function is invoked. Its return value is ignored. The `code_change` function is used to update the internal state in case of an code hot-swap, an Erlang feature which is not covered in this work.

The `Responder` as tail-recursive process is an extension of the ping-pong server in Figure 2.1. We extend the above shown process to respond to messages in an infinite loop. These kinds of processes are called *long-running*.

There are no actual loops supported in Erlang, but instead one implements equal behavior with recursive calls. If these calls are the last statement in a function, they are called *tail-recursive*. In that case the invoking (or callers) stack frame can be discarded, as no return to this function can happen.

In Figure 2.3, we show the implementation of the `Responder` as tail-recursive process, which in the same manner as the `gen_server` example above stores a list of processes, it has replied to. It can maintain this list, by having a `State` variable, which is updated with `New_State=[Other,State]` in every recursive invocation, shown in Lines 2–4. The updated state is then passed on to the next invocation. The tail-recursive call can be seen in Line 6. In Lines 7–9, an anonymous function, which initializes the `loop` with an empty list, is defined and can be used in the same way as in Figure 2.1.

```
1 loop (State) ->
2   New_State=receive {Other,ping} ->
3     Other!pong,
4     [Other,State]
5   end,
6   loop(New_State).
7 Responder_fun=fun()->
8                 loop([])
9               end.
```

Figure 2.3.: A tail-recursive ping-pong example

2.2. Abstract Behavioural Specification

The ABS language[1] was developed as part of the EU FP7 HATS (Highly Adaptable and Trustworthy Software using Formal Models) project to allow the modeling and analysis of adaptable distributed concurrent software. Alongside a toolsuite to support development was created, which provides an Eclipse plugin, Unit testing, package/dependency management and visualization tools [WAM+12].

ABS is an object-oriented language with functional expressions on a statement level. Concurrency is enabled by providing asynchronous calls and cooperative scheduling of those calls in groups of objects. Furthermore, components can be reused and newly combined to adapt modern software to the ever changing requirements it has to fulfill.

Cooperative scheduling is a concept on how a set of executable units manage to share a single execution slot, where one unit is executed. When scheduling cooperatively the current executing unit has to willfully give up its computation, to allow another unit to be scheduled. This is called to *suspend* in ABS. In such a system a unit cannot be forced to pause its computation. Therefore all units have to cooperate to achieve the desired system behavior.

The language is split into two levels: The Core ABS language [JHS+10], which contains functional, sequential, object-oriented, concurrent and distributed concepts to describe a single model. The syntax is inspired by the Java language where applicable. The second level is Full ABS, which allows the description of variable composable models via delta-oriented programming [SBB+10] and software product lines [PBvdL05]. One composition of variable model parts is called a product. A single product, selected by a set of features and generated by Full ABS results in a model only containing Core ABS elements. So one merely has to consider Core ABS models for execution. Therefore, this work focuses on Core ABS and will discuss that in detail and than give a short overview about Full ABS.

[1] http://abs-models.org

2.2.1. Core ABS

ABS supports algebraic data types and first order functional expressions. They can be composed with sequential and object-oriented constructs. Concurrency modeling capabilities are added on top of object-oriented concepts. This layered design supports formal reasoning on models, as lower layers are less complex and therefore easier to reason about. Like the functional layer, which is side-effect free and has immutable state. We now have a look at all these layers and how to compose them to form a powerful modeling language. More information can be found in a tutorial by Hähnle [Häh12] or the current language documentation under `http://docs.abs-models.org`

Built-in Types and Algebraic Data Types ABS has besides the common built-in types `String`, `Int` and `Bool` also the following types:

`Unit` : has only a single value, used if no value should be returned from a method
`Rat` : represents a rational number, with arbitrary large numerator and denominator
`Fut<A>`: a future containing potentially a value of type `A` (see the paragraph on asynchronous calls on Page 16)

```
data List<A> = Nil |
               Cons(A head,
                    List<A> tail);
def Int length<A>(List<A> list) =
  case list {
    Nil => 0 ;
    Cons(p, l) => 1 + length(l) ;
  };
```

(a) Functional parts of ABS

```
{
  ListServer dl= new local
      DuplicatingList(42);
  dl.add();
  Fut<Int> res= dl!length();
  await res?;
}
```

(b) Main block with concurrent parts of ABS

```
module Lists;
export ListServer, DuplicatingList;
interface ListServer {
  Unit add();
  Int length();
}
class DuplicatingList(Int value)
    implements ListServer{
  List<Int> store;
  {
    store= Cons(value,Nil);
  }
  Unit add(){
    store= Cons(value,store);
  }
  Int length(){
    return length(store);
  }
}
```

(c) A module with a class and an interface

Figure 2.4.: Example language constructs in ABS

Algebraic data types allow the specification of composed immutable values, using constructors and other data types. Those can also be parametric typed as can be seen in Figure 2.4(a) Lines 1–3, where a list is defined over the parametric type `A`. This list type can either contain the empty constructor `Nil` or a `Cons` cell over an element of `A` and a `List<A>`.

Functional Expressions and Functions Next to standard side-effect free expressions like **let**, **if** and function application, ABS supports pattern matching via the **case** expression. An example can be seen in Figure 2.4(a) Lines 5–8, where one list cell is matched against two different patterns. The second pattern `Cons(p,l)` contains the not previously used variables `p` and `l`, which will be bound in case of a match. When evaluating a **case** expression, the expression on the right-hand-side of the arrow `=>` of the first match will be evaluated for the value. A functional expression can be defined as a function like shown in Line 4 of the Figure 2.4(a) .

Sequential Programming Next to expressions, which can be evaluated to a value, ABS supports statements like assignments, variable declarations and other control flow statements like **if**, **case** pattern matching, **while** loops and blocks of statements. They can be composed to form the method's body.

Interfaces and Classes To support object-oriented programming ABS supports classes and interfaces. Interfaces support multiple inheritance of other interfaces and classes can implement multiple interfaces. Only interfaces are used as types of objects (instead of classes). Subtyping for derived interfaces is supported. As object references are only typed on interfaces one can also only refer to methods and not to fields. There is no class inheritance in ABS. A different approach for specialization is provided by supporting delta modules (see Full ABS). This leads to the fact that no member visibility specifications are required as everything in an interface is public and in a class private.

Classes do not have normal constructors, but instead support class parameters, which represent the values that have to be passed to a newly created object. Additional initialization code can be placed in an optional init block. Furthermore, classes can be *active* as proposed by Johnsen et al in [JOA03]. The optional method, named `run`, will be executed after instantiation to represent the active behavior of an object.

The example in Figure 2.4(c) shows the definition of an interface with two methods over Lines 3–6. The interface `ListServer` is implemented by the class `DuplicatingList`, which maintains a list of integers. Each of these list elements has the same value. This value is set via a class parameter (`Int value`). An init block (Lines 9–11) initializes the internal list with a single element. The implemented methods make use of functions and data types defined in Figure 2.4(a). No import statements are needed for them, as they are part of the standard library.

Modules and the Main Block The module system allows to structure code elements similar to the one found in Python. A module consists of definitions of functions, classes, interfaces, algebraic data types and imports from other modules. Furthermore, one can define via exports, which definitions are available to other modules. A module can also contain a block of statements, the so-called main block, which will be executed after the startup of the model.

Concurrent Object Groups and Tasks/Processes Concurrent Object Groups (COGs) are one central aspect how concurrency can be handled in ABS. They represent the smallest unit of distribution in ABS. Following the concept of *CoBoxes* by Schäfer et al. [SP10] a COG is a group of objects with a set of tasks that are processed by those objects. Only one of the COG's tasks can be execute at a time and therefore, as all fields are private, there can be no concurrent access to the objects in a group.

A task represents the execution of sequential code in an object and can either be:

- An active object's `run` method
- An asynchronous call (see below)
- Execution as part of the main block

In between objects of a COG, calls can be performed synchronously, whereas intra-COG calls have to be asynchronous. COGs do not have a syntactical representation in the ABS language, so they are not themselves directly visible as a grouping mechanism in a model. There is also no COG type that one could refer to. Instead one can control, if a new object is placed in a new COG or in the object's creator's COG by using either **new** `<Class>` or **new local** `<Class>` construction operation.

In Figure 2.4(b) a main block is depicted that uses the above explained class and interface. In Line 2 we instantiate a new object, which will run in the same COG as the main block. Therefore, we can use a synchronous call to the `add` method in Line 3.

Tasks are also called processes in ABS-related literature. We choose to stick with the slightly less used term task, because ABS tasks and Erlang processes appear throughout this work in the same context, which could easily lead to confusion.

Asynchronous Calls and Cooperative Scheduling Asynchronous calls are not only necessary to invoke methods on objects in different COGs, but also allow to make multiple calls concurrently. An asynchronous call will return immediately a future [dBCJ07], which is a container for the result value of this call. Unless the result is already contained in the future, the future is unresolved. In ABS the result of a future `f` can be accessed with the **get** expression. In case the future is still unresolved, the **get** expression blocks until the future resolves. Thus an asynchronous call with an immediate **get** on the future can be used to simulate a synchronous call.

By using cooperative scheduling ABS offers a way to handle concurrency in a COG, which allows simplified reasoning over models and avoids the need of locks or mutexes. At explicit scheduling points the execution in the COG can change from one task to another. There are two different statements for that in ABS: the unconditional **suspend** and the **await** `C`, where a task suspends and is only allowed to resume if the condition `C` holds. A condition can either be an expression that evaluates to a `Bool`, the operator `?` that can be applied on a future and resolves to true if the future is resolved, or a conjunction of two conditions with the operator `&`.

The combination of **await** and asynchronous calls allows us to implement concurrent systems, where the **await**s can function as synchronization points. This is depicted in Figure 2.4(b), where in Line 4 the asynchronous call is performed and in Line 5 an **await** on that future happens.

How cooperative scheduling with **await** statements can ease the specification of concurrent systems, can be seen in Figure 2.5(a). In the example a buffer is shown, which has a fixed `capacity` and stores the data in an internal list. It allows concurrent calls to the `read` and `write` methods, which as the name indicates put or retrieve a value from this buffer in First In First Out (FIFO) order. The **await** statements in Lines 6 and 13 help to guarantee that the methods only continue, if the buffer is either non-empty or still has available capacity.

The ABS model is here also compared with an implementation in Java, depicted in Figure 2.5(b). We can guarantee mutual exclusive access by making the `read` and `write` methods `synchronized`. The conditions, which are checked with **await**s in ABS, are implemented by a `while` loop, `wait` and `notifyAll` invocations. A condition is checked in the `while` loop and in case it does not hold the `wait` method is invoked, as can be seen in Lines 7–9 and 18–21. The `wait` method leaves the monitor, so allows another thread to enter a `synchronized` block, and resumes if it is notified by a `notifyAll` invocation. The `notifyAll` calls have to be placed, wherever a waiting thread's condition could have changed, which is in our case after a `read` in Line 13 or after a `write` in Line 24.

Annotations are additional markup that can be placed in front of all definitions of functions, classes, interfaces and type usages. They allow to easily extend the modeling, by providing additional information, without the necessity to implement a new special syntax. A single annotation is composed of a side-effect free expression and an optional type specification.

For example an object can be in the same or a different COG, so the references to this object are tracked in the compiler as *near* or *far*, to ensure that no synchronous calls are attempted on a *far* reference. Normally this information is inferred by the compiler. In cases where this is not possible, like when an object is created outside of a method's scope, one can annotate these references with **[**`LocationType: Far`**]** or when omitting the type specification with **[**`Far`**]**.

```
class IntBuffer{
  Int capacity= 10;
  List<Int> buffer=Nil;

  Int read(){
    await ~isEmpty(buffer);
    Int val= head(buffer);
    buffer= tail(buffer);
    return val;
  }

  Unit write(Int val){
    await length(buffer)<capacity;
    buffer= appendright(buffer,val);
  }
}
```

(a) A buffer in ABS

```
public class IntBuffer{
  private int capacity= 10;
  private List<Integer> buffer=
              new ArrayList<>();

  public synchronized int read(){
    while(buffer.isEmpty())
      try{
        wait();
      }
      catch(InterruptedException e){}
    int val=buffer.remove(0);
    notifyAll();
    return val;
  }

  public synchronized void write(int
      val){
    while(buffer.size()>=capacity)
      try {
        wait();
      }
      catch(InterruptedException e){}
    buffer.add(val);
    notifyAll();
  }
}
```

(b) A buffer in Java

Figure 2.5.: A comparison between two concurrent buffer implementations

2.2.2. Full ABS

Modern software needs to be adapted to a wide range of uses by potentially different customers. Therefore, one wants to reuse components, which can speed up and improve the development process. Reusabilty and variability can also be beneficial when modeling and testing a system, as it allows to test different software components. Full ABS implements the concept of software product line engineering [PBvdL05, CMP+10], allowing for this kind of modeling.

A product is a Core ABS model, which fulfills a set of features. Features can be modeled by applying a set of changes to a minimal model. These changes, like adding or changing methods, adding types and classes, are expressed as delta modules. An example for a delta module can be seen in Figure 2.6. In Line 26-31 a delta is defined that modifies the `sayHello` method of the `Greater` class. Another delta is shown in Line 7-19, which also changes the same method, but calls via **original** the version of the method before applying

```
1  module Hello;
2  class Greeter implements Greeting {
3    String sayHello() {
4      return "Hello world";
5    }
6  }
7  delta Rpt (Int times);
8    modifies class Greeter {
9      modifies String sayHello() {
10       String result= "";
11       Int i= 0;
12       while(i < times) {
13         String orig= original();
14         result= result + " " + orig;
15         i     = i + 1;
16       }
17       return result;
18     }
19   }
```

```
20 delta De;
21 modifies class Hello.Greeter {
22   modifies String sayHello() {
23     return "Hallo Welt";
24   }
25 }
26 delta Nl;
27   modifies class Greeter {
28     modifies String sayHello() {
29       return "Hallo wereld";
30     }
31   }
32 productline MultiLingualHelloWorld;
33 features English, German, Dutch,
       Repeat;
34 delta Rpt(Repeat.times) after De, Nl
       when Repeat;
35 delta De when German;
36 delta Nl when Dutch;
37 product P1 (English);
38 product P2 (Dutch,Repeat{times=10});
```

Figure 2.6.: Multilingual Hello World shipped with the ABS frontend found under `tests/abssamples/deltas/Hello.abs` licensed under the Modified BSD License

the delta. This highlights how multiple deltas can be stacked, but also the importance of the order of the application. Furthermore, the second delta also has a parameter `times`, which allows for build-time parameterization of ABS models.

All deltas necessary for applying a feature, are specified by a **productline** definition as can be seen in Figure 2.6 Line 33-36. First all available features are listed, which are in our example: `English`, `German`, `Dutch` and `Repeat`. With the **when** keyword a list, which defines when it should be applied, is specified for each delta. Optionally also the order can be configured with the **after** clause.

To generate a product we first specify different kinds, like the products `P1` and `P2` in Figure 2.6 Line 37-38, where the part in the brackets represents the selected features with parameters. Such a product can be built at the code generation step by adding a parameter `-product=P1`.

In the shown model we could also specify a product that would have the feature `dutch` and `german`, which is kind of nonsensical in this example as this program can only greet in one language. Therefore one can also specify a feature model, that allows to put constraints on the set of selected features and their parameters. For details see the reference manual or the case study by Helvensteijn et al. [HMW12].

Chapter 3.

Case Study

In the following pages we present the case study this work is built upon and its requirements. Then we show how this case study can be modeled in ABS and how its concurrency model allows to fulfill the requirements.

3.1. The Video Transcode Server

The Video Transcode Server (VTS) is a facility, allowing a user to request a video file, which is stored in a codec known to the server. It will be on the fly encoded in a codec known by the client. The server could have more codecs available, which enable the clients to receive files, they normally would not be able to decode.

A video consists of a sequence of frames, which only can be consumed linearly, so no jumps or reverse playback is supported. The following terminology will be used to describe the encoding in the server: a `Block` refers to one encoded `Frame` of a video and is one element of a `BlockStream`, which represents the linear access to the video file.

The communication between both parties is quite simple: The client requests a file by sending its name to the server. In response the server sends transcoded blocks to the client until the end of the file is reached. To maintain simplicity, the used codecs are hardcoded in this version.

The server has multiple requirements:

- it should be able to serve multiple clients
- as encoding is expensive, only those parts of a video file that will be retrieved by the client should be processed
- to enable continuous lag-free playback some transcoded frames should be precomputed and cached before they will be retrieved

As encoding, decoding and file access go beyond the scope of this work and a model in general, they will be only simulated by resource consuming functions.

Connection model Further simplifications will be used for modeling the server-to-client communication. We want to have a model, that can be fully represented in ABS without the use of calls to other programming environments or operating system routines, which could provide a real networking stack. The communication of a client and a server is modeled by a connection object, synchronizing via asynchronous send and retrieve calls. In Figure 3.1 one can see the interfaces and classes used to model a connection. The `SyncConnection` class represents the connection, which is accessed either via the `ServerConnection` or `ClientConnection` interface, depending on the usage side.

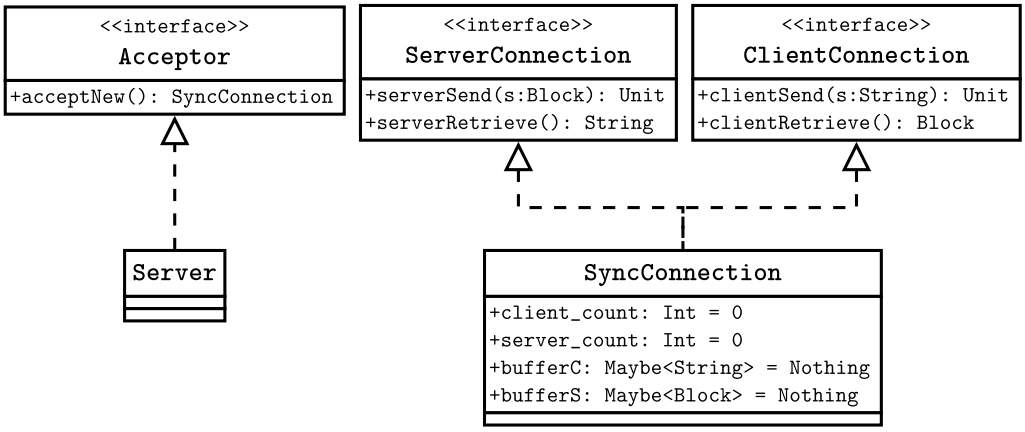


Figure 3.1.: Classes used to model a typed bidirectional synchronous connection

A connection is established by calling the `Acceptor`'s `acceptNew` method, which is implemented by the `Server`. This method will create a `SyncConnection` object in a new COG and return the object to the client. On the server-side it is passed along to a newly created `ConnectionHandler`, which performs the communication with the client. This simulated bidirectional synchronous connection has fixed data types for sending and retrieving in the following way: a client can only send messages of type `String`, the server can only send `Blocks`.

For instance, server to client communication through this `SyncConnection` object is done, by the server invoking `Unit serverSend(Block s)`, which will wait until the client called `Block clientRetrieve()`. The order could also be reversed, as both calls perform a barrier synchronization by using **await** statements. Barrier synchronization means that a group of executable units (here objects) waits at a barrier and they can only continue, after all executable units have reached this barrier. In the connection the barrier is represented by the pair of `send` and `retrieve` methods.

Implementation of the Connection The `SyncConnection` object provides two independent channels – one for each direction. As they are conceptionally equivalent we describe how the connection from the server to the client works.

The implementation of the operations `serverSend` and `clientReceive` for the channel from the server to the client is depicted in Figure 3.2. This channel sends data of the type `Block`. It requires two fields: a `server_count`, representing how many messages the client has consumed, and a `bufferS`, which can contain a single message or is otherwise empty. The latter is implemented with the `Maybe<T>=Just(T t)|Nothing` type shipped with the standard library, where the functions `isJust` and `fromJust` return in case it is filled `true` or respectively the contained data.

```
1 Unit serverSend(Block msg){
2   await bufferS==Nothing;
3   bufferS=Just(msg);
4
5   Int oldVal = server_count;
6   await server_count == 1+oldVal;
7 }
```

(a) Send

```
1 Block clientRetrieve(){
2   await isJust(bufferS);
3   Block msg= fromJust(bufferS);
4   server_count  = server_count+1;
5   bufferS   = Nothing;
6   return msg;
7 }
```

(b) Retrieve

Figure 3.2.: Operations on a single direction connection

The `serverSend` method has to ensure that the previous content was read from the `bufferS` before it fills the `bufferS` and waits until the `server_count` is incremented by one, which indicates the read from the client. The `clientRetrieve` method just has to wait until the `bufferS` is filled, retrieve the content, increment the `server_count` and empty the `bufferS`.

The connection model is also depicted in the sequence diagram in Figure 3.3. In that example a client opens a connection and sends one message to the server-side, which is represented by the `ConnectionHandler` object.

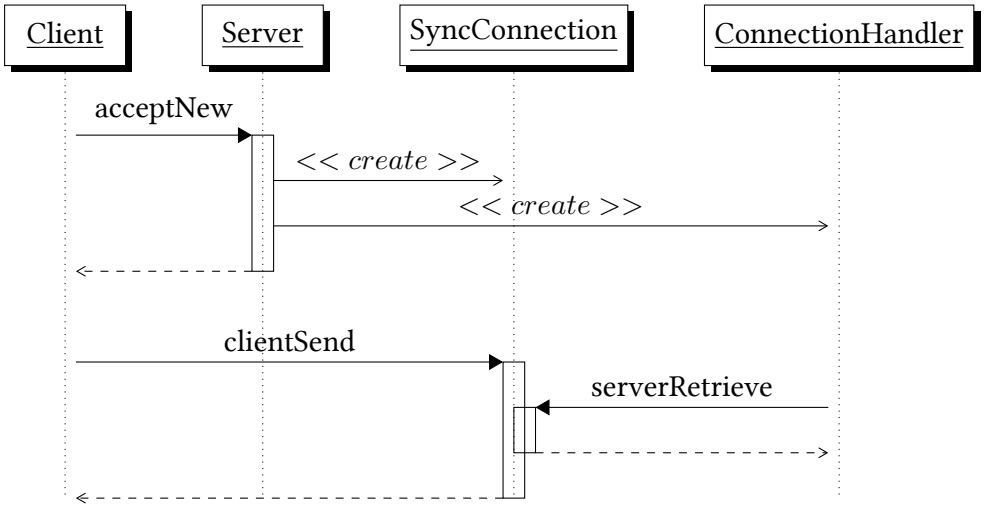


Figure 3.3.: Messaging in Video Transcode Server

Usage of the case study in this work The Video Transcode Server will be implemented and adapted throughout this work. In the next section it will be modeled in ABS, while especially considering the above stated requirements. After we have introduced the new error handling primitives, we analyze possible runtime errors and extend the model to handle those in Section 6.5. In Section 6.6.2 we then implement a supervised global pool of encoders to regulate the amount of parallel encoding, while still ensuring an error-tolerant system.

3.2. Modeling the Case Study in ABS

The above introduced case study will be translated to an object-oriented model for ABS. The classes and interfaces of the server are depicted in Figure 3.4. Those and other design decisions will be explained in the next paragraphs.

Each object is in its own COG to enable concurrent execution. The `Server` class, which is depicted in Figure 3.1, enables the `Client` to retrieve a connection by calling its `acceptNew` method. This connection, which is, as described on the previous page, represented by an object, is observed by a `ConnectionHandler` object. It is created by the server for each connection and will in its active behavior await an initial message from the client and then fill the stream with transcoded blocks from the requested file. In order to produce the transcoded blocks, the connection handler will create an instance of the `Transcoder`. After the initialization transcoded blocks can be retrieved by calling `nextBlock()`.

A `Transcoder` can use realizations of the interfaces `Encoder` and `Decoder` to fulfill its transcoding task. Those interfaces are defined in a very abstract manner to enable implementations supporting various codecs. Furthermore, the input is abstracted by using the `BlockStream` interface, which provides methods to check if more input is available and retrieve it. The three interfaces `BlockStream`, `Decoder` and `Encoder` are together referred as the *pipeline*, because blocks pass through these elements sequentially and will be transformed in this process. The end product of the pipeline's application is a sequence of differently encoded blocks. No implementation of the interfaces is depicted, as we want to focus on the architecture of the server and the *pipeline* in general. For testing we use dummy implementations for each of the interfaces.

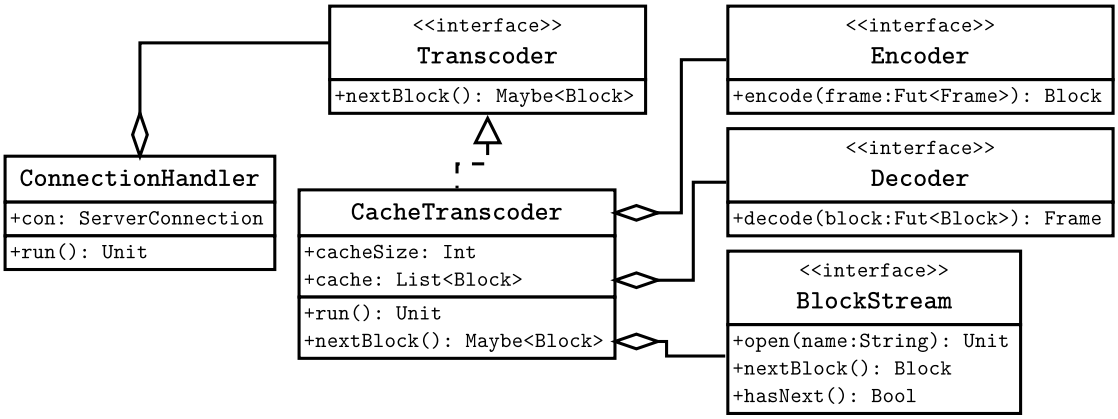


Figure 3.4.: Class diagram of the transcoding server

The CacheTranscoder is the realization of a `Transcoder`, which implements the desired properties, that blocks should only be encoded on demand and that a cache of the next few blocks should be maintained. An instance must be initialized with an object for each of the

pipeline elements (see previous paragraph). The `CacheTranscoder`'s active object behavior will try to cache a configurable amount of blocks. In between this effort to fill the cache, the transcoder also has to serve the `Client`, by processing `nextBlock()` invocations. Both goals can be easily achieved by using asynchronous calls and **await** statements.

Next, we discuss in detail the `run` method shown in Figure 3.5a. In a loop the `run` method tries to fill the cache. In order to maintain a fixed cache size, the whole loop executes only if the first **await** in Line 3, which checks that the cache size is smaller than the fixed maximum size, holds.

To handle the scenario, when the `BlockStream bs` does not provide further blocks, the behavior's main loop has a `shutdown` flag. In the check in Lines 4–8, the `BlockStream`'s `hasNext` method is called and if no more blocks are available, the `shutdown` flag is set to `True`. This stops also the loop in Line 2, which in turn ends the active behavior.

Looking at the normal mode of operation, one sees the caching behavior over the Lines 10–15. The *pipeline* is not strictly built one pipeline stage after the other. Instead all steps are called asynchronous and the necessary result of the previous step is passed to the calls as a future. This architecture is built in the Lines 10–13 and leads to the following behavior. A pipeline operation can perform some precomputation, when invoked (e.g., some pattern analysis of the previous frames) and only synchronizes with the previous step when accessing the future's value. In case that is not computed yet, the access operation will block. This behavior is also depicted in Figure 3.6, where all three operations are shown over the time of transcoding one frame. Blue phases show precomputation. In red phases

```
1  Unit run(){
2    while(~shutdown){
3      await length(cache) < cacheSize;
4      Fut<Bool> fNext= bs!hasNext();
5      await fNext?;
6      Bool next= fNext.get;
7      if(~next)
8        shutdown= True;
9      else{
10       Fut<Block> block = bs!nextBlock();
11       Fut<Frame> frame = d!decode(block);
12       Fut<Block> result= e!encode(frame);
13       await result?;
14       Block store= result.get;
15       cache= appendright(cache,store);
16     }
17   }
18 }
```

(a) `run`

```
1  Maybe<Block> nextBlock(){
2    await ~isEmpty(cache) ||
         shutdown;
3    Maybe<Block> retval= Nothing;
4    case cache {
5      Cons(x,xs) => {
6        cache = xs;
7        retval= Just(x);
8      }
9      //Retval stays Nothing
10     Nil => skip;
11   }
12   return retval;
13 }
```

(b) `nextBlock`

Figure 3.5.: Key methods of the `CacheTranscoder`

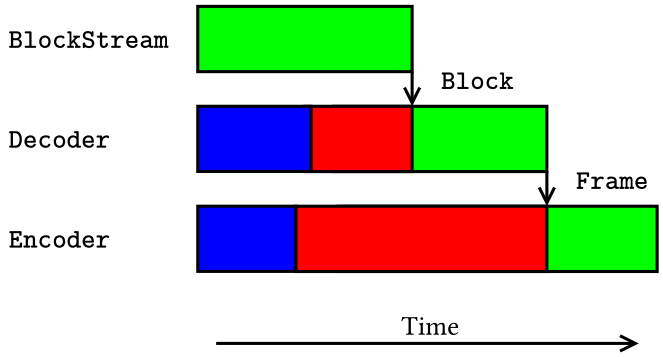


Figure 3.6.: Pipeline execution phases over time

a operation is blocked by waiting on the previous result and in green phases the computation involving the data happens. At the end of a green phase, the arrow indicates the data, that is passed on, by filling the future.

To retrieve the transcoded frame the `CacheTranscoder` waits in Line 13 until the future of the last pipeline step is completed. In the following lines the value is retrieved and stored in the cache.

The `CacheTranscoder`'s `nextBlock` method is depicted in Figure 3.5b. It retrieves the next transcoded block from the cache. An **await** statement ensures that either the cache is not empty or the shutdown flag is set. In case of a shutdown, no more frames will be produced so waiting for an non-empty cache is nonsensical. After continuation at the **await** a pattern match retrieves and removes the first cache element or results in a special `Nothing` term, if the last block was already retrieved. This value is then returned.

How values are passed from one object to another in the Video Transcode Server, can be seen in Figure 3.7. The blue arrows are directed from a value's source to its accessor. For instance, the `ConnectionHandler` accesses the return value of the `CacheTranscoder`'s `nextBlock` method and the result of the `SyncConnection`'s `serverRetrieve`.

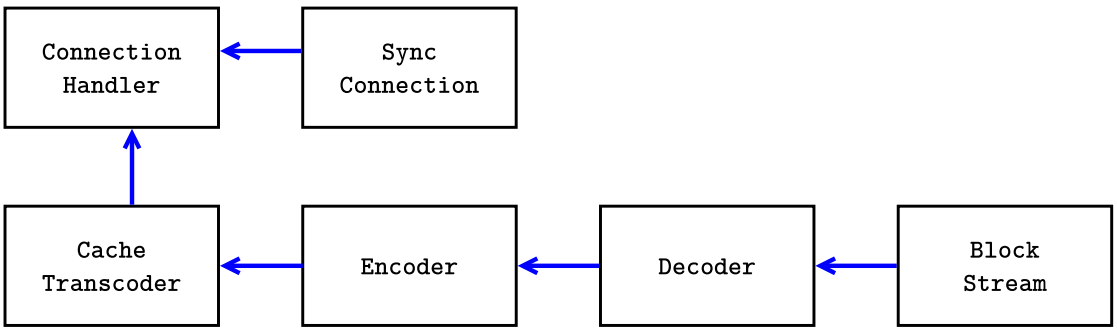


Figure 3.7.: Value propagation in the Video Transcoder Server's Objects

We have shown, that the `CacheTranscoder` provides concurrent filling and accessing of the cache, by the use of simple **await** statements. The complete listing of all modules can be found in the Appendix A.

Chapter 4.

Mapping of ABS to Erlang

In this chapter we present a concept to translate and execute an ABS model to Erlang code [GAJ+13]. To generate this code it is important to determine how to represent basic ABS constructs, like objects, COGs, futures and tasks in Erlang. One has to consider how their structure looks at execution or runtime. Furthermore, a static code organization view dealing with ABS classes, modules and functions has to be considered as well.

First, we present the translation concept, then we look closer into the runtime, which provides a framework for the generic components of translated model. The chapter closes with an example, which highlights how all the different Erlang processes interact and which messages they exchange.

4.1. Translation Concept

First ideas Different ideas how an ABS program can be translated come to mind. We will discuss some of them.

One approach would be to have one Erlang process per COG. This process would execute a single task at a point in time and hold the state of all objects and tasks. This is motivated by the simplicity in scheduling for the Erlang runtime and that there would be a small number of processes, which are in an executable state for most of their lifetime. The concept and advantages/disadvantages are quite similar to an user space multithreading solution for an operating system.

This idea would require a relatively complex logic in the COG process. Furthermore, one has to store all objects states and tasks with their stacks in one COG. Scheduling would still be needed to be implemented because the COG process has to select and execute a task. As advantage can be seen that this one process has a full and consistent view of the whole COG state, but it also leads to one very complex process, which should be, following Erlang principles (see Chapter 3 of [Arm03]), be avoided. Moreover it also limits the possibilities to use features provided by the Erlang runtime, like linking and process independence, which in later stages of this work will be used to introduce new concepts.

Chosen approach A different concept, which seems more appealing, could be summarized as, *everything is a process*. The core idea is to structure all key elements of a software in self-contained processes that communicate with each other. This design's benefits are: a clear communication structure, as a process's data is encapsulated; isolation of errors in the processes, with the ability to observe such if needed. The choice of this key elements is similar to what one would choose as classes for an object oriented language, while omitting those classes, that are only used to structure data. In general, the granularity of processes is a design choice and has to be guided by experience. For the translation, we chose to represent objects, COGs, tasks and futures by processes.

By using the *everything is a process* concept, we hope to reach a better code/data separation in multiple modules and processes, leading overall to smaller and simpler components. The basic runtime entities will be represented in the following way:

Object: a process holds the object's state (all member variables). Field access is performed by synchronous messaging.

COG: a process manages all tasks and schedules them by handing around a token via messages. Further it has to keep track of all tasks' execution state.

Task: a process representing one execution unit, so either a main block, a remote object initialization, an active object's run method or asynchronous method invocation.

Future: a process, which is on one side the synchronous execution part of an asynchronous call in the called object's COG and on the other side it is the container for the result of the execution. As part of that it has to support the access and waiting of other process on the result, which is once again done via messaging.

Figure 4.1 shows a graphic mapping of ABS components and references between them to Erlang processes. In this illustration two objects in a COG are represented as Erlang processes. The sample objects are taken from the Video Transcode Server case study. Dotted lines show which component transforms to what Erlang counterpart, where solid lines with arrows above show a reference from one entity to another. On the ABS side one can see the active `CacheTranscoder` referring to the `Encoder`, which in Erlang is translated to a `COG` process, which references a `Task` process executing the `run` method. The task itself has a reference to the object (`CacheTranscoder`) it is working on. All necessary references can be represented by the PID of the target process in Erlang.

Code organization There exists not only this dynamic or runtime scoped view, but also a static view on how to translate ABS to Erlang. The main idea is to represent a class by one code module. The constructors and the methods of a class are each translated into a function, which is invoked with a reference to the called object as first parameter. For example, the method `serverSend(Block block)` of the class `SyncConnection` is represented by the function `m_serverSend(Obj,V_block)` in the module `class_SyncConnection`. The `Obj` is a parameter, that is bound on the invocation to an object of the class `SyncConnection`. One can also see that all names need to be prefixed when translated to Erlang, because the

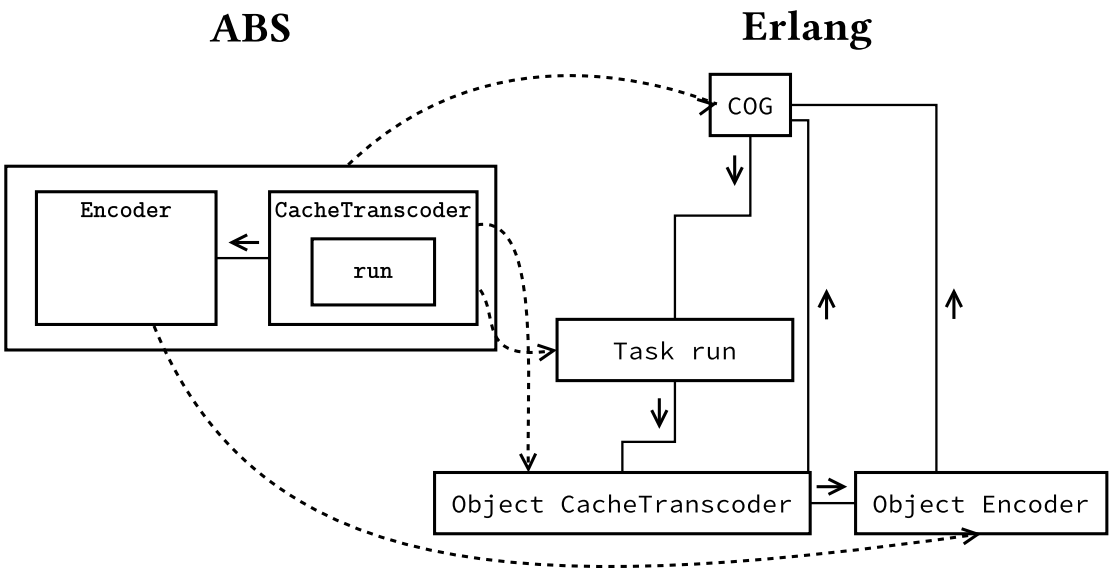


Figure 4.1.: Dynamic view on a sample translation

capitalization determines the difference between types or function names (or more generally atoms) and variables. Furthermore, the clashing of internal used names and ABS names can be avoided by prefixing. The namespaces built-up by ABS modules are also part of the prefix of classes and functions.

At first it was considered to model method visibility – public methods in ABS are specified in interfaces – by only exporting these public methods by a class module. As asynchronous calls in ABS can also be invoked on private (non-interface) methods, those need to be externally accessible as well. Therefore visibility is not modeled in the Erlang backend and all functions are exported. One has to note, that this does not allow a faulty model to execute calls to internal methods, as such would be rejected by the compiler frontend.

Due to Erlang's dynamically typed nature, interfaces are ignored in the backend, but the conformance of interfaces and classes is checked by the frontend.

In addition to classes and interfaces an ABS module can also contain pure functions and a main block. All functions are generated into a special module `m_moduleName_funs`, while a main block is put into a `main` function in an Erlang module called like the ABS module with an additional prefix. This main method can then be started as `main_task`.

The runtime, with code for all the above mentioned processes and utility functions, is also organized in separate modules. As concrete tasks or object modules have some common functionality (like scheduling callbacks or object initialization), they are implemented as behaviors. This OTP concept splits a problem in a process with generic functions and a callback module with a defined set of functions, modeling the custom behavior of each

realization. In the runtime the actual task execution, object initialization or field access code is part of a callback module, whereas management or the process execution loop can be implemented in a general fashion.

In Figure 4.2 a translation of the ABS class `CacheTranscoder` with its methods and attributes to the equivalent Erlang module is shown. In the `class_CacheTranscoder` module, we see additional functions, besides those representing the methods `nextBlock` and `run`. With the `init(Obj,[P_cacheSize])` function the object is initialized, the `P_cacheSize` parameter represents the class parameter for the field `cacheSize`. The callback functions `get_val_internal`, `set_val_internal` and `init_internal` are part of the mechanism to handle object fields, and are described in more detail in Section 4.2.

Furthermore, the most important runtime modules with their exported functions are visualized. They cover the four kinds of processes, presented above. With the `<<behavior>>` stereotype we highlight the modules, which are implemented as Erlang behavior and require a callback module. For the `object` module, the `class_CacheTranscoder` is such a callback module. Details of the runtime modules are discussed in Section 4.2.

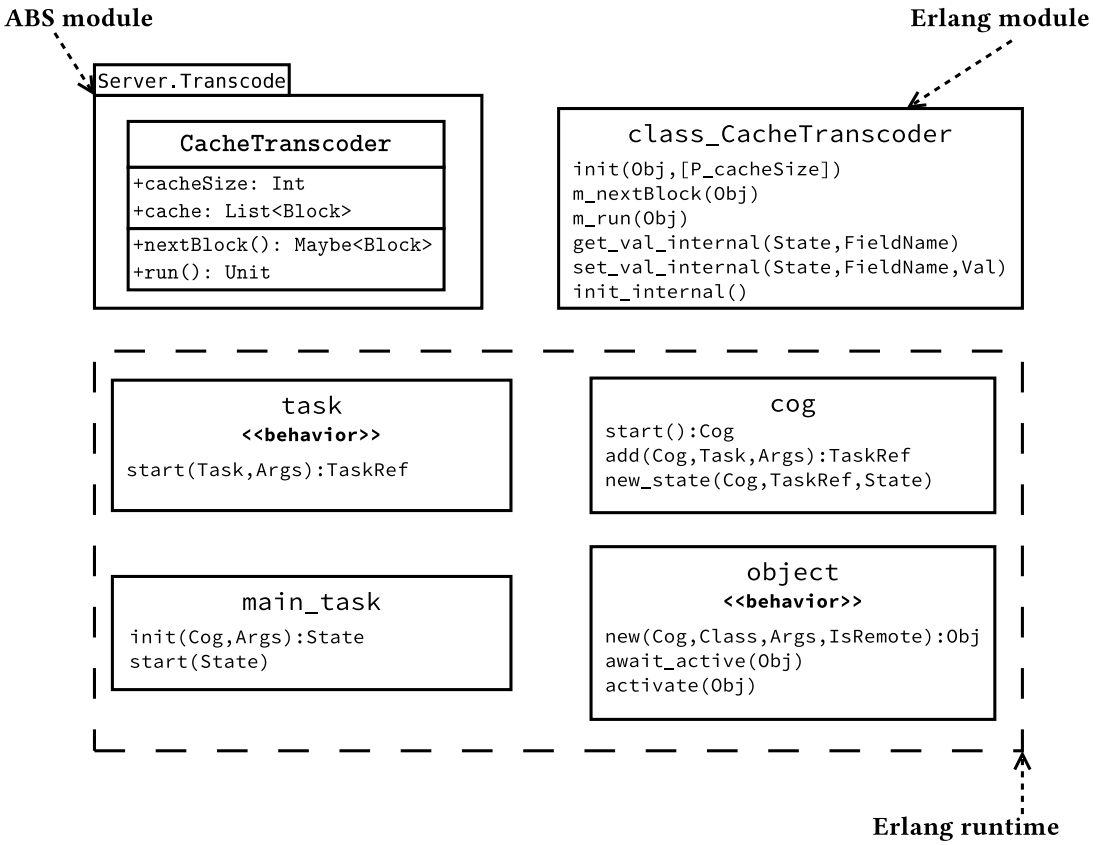


Figure 4.2.: Static view on a sample translation

Data types The ABS data types can mostly be translated to data types available in Erlang. `Bool`, `Int` and `String` are mapped to their equivalent Erlang type. As there is no built-in data type to represent rational numbers, the `Rat` type has to be handled separately (see the paragraph on rationals on Page 50).

Algebraic data types in ABS are built out of *data constructors*. Those have a value describing the type (*TypeId*) and optional parameters. Constructors that take parameters are represented by a tuple, where the first element is the *TypeId*, followed by all other parameters as elements. A constructor without parameters is instead represented directly by its *TypeId*. All *TypeId*s, which are not built-in types, are represented as an atom with the literal value of the *TypeId* and the prefix `data`. An example of this can be seen in Figure 4.3, where we show the definition of the type `Set`, an empty set `Set1` and a set `Set2` containing the values 2 and 3. On the left hand side the ABS code and on the right hand side the Erlang translation is shown. We also see that the type definition is omitted on the Erlang side.

```
1 data Set<A> = EmptySet |
2               Insert(A, Set<A>);
3 Set1=EmtpySet;
4 Set2=Insert(2,Insert(3,EmptySet));
```

(a) ABS set definition and ex

```
1 Set1=dataEmptySet,
2 Set2={dataInsert,2,
3       {dataInsert,3,dataEmptySet}}
```

(b) Example sets in Erlang

Figure 4.3.: The type `Set`: definition and a translation of values

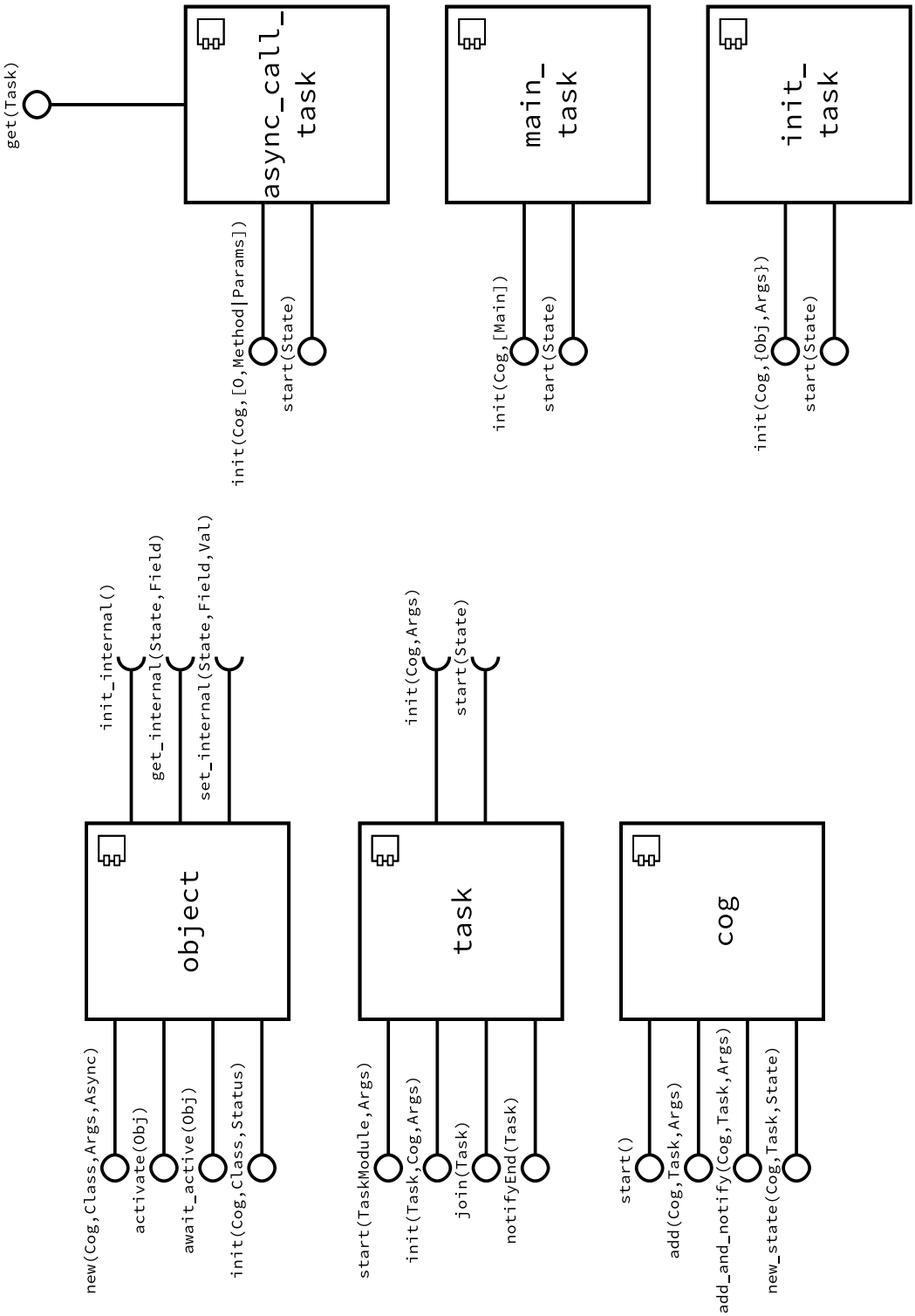


Figure 4.4.: Overview of runtime components

4.2. ABS entities in Erlang

The Erlang realizations of the fundamental ABS entities *object*, *COG*, *task* and *future* will now be described in more detail. A translated model is split into two parts: a static runtime that provides generic functions and processes, and the part that is specific to the model like its functions and classes. The objects of the classes are represented by using the Erlang behavior pattern. With this pattern a general object process is implemented, and a class module, acting as behavior module, has only to provide callbacks, which implement the access to the object's fields.

In Figure 4.4 we give an overview of the most important runtime modules by using a component diagram. The modules are depicted as rectangles, with provided functions as circles, and expected callbacks in a behavior as semicircles.

In the overview, we also see that the `task` module is implemented following the behavior pattern, where the specific behavior for the different kinds of tasks is implemented by callback modules. All the shown modules will be discussed in detail over the following pages.

Object An object is realized through an Erlang process, that will run in a tail-recursive loop and handles the object's state. The general concept is explained in Section 2.1 in the paragraph *The* `Responder` *as tail-recursive process* on Page 11. The object process accepts messages to retrieve and set values. The handling of member fields is done by calling the callback module's `get_val_internal`, `set_val_internal` and `init_internal` functions. They return a state term, which is passed to the next invocation of one of those callback functions and is stored by passing it along in the object process's tail-recursive loop.

Those callbacks are generated in the class module and represent the state as a record, where every member variable and class parameter is a field.

New objects are created by the function `object:new(Cog,Class,Args,Async)`, where the `Args` parameter is a list of arguments that are passed to the init block of this class. If an object is created on a COG different from the caller's COG, the last parameter has to be `true`. In that case the object is not directly initialized, but this is done via an initialization task on the remote COG. Therefore, the `init` function, which is the starting point of a newly created object process, has to be exported as well.

Furthermore, a mechanism is needed to check if the object is already initialized and can be used. Otherwise an asynchronous call to an uninitialized object could be executed. By calling the function `object:await_active`, the callee will block until the invocation of `object:activate`, which is performed at the end of the initialization.

```
new(Cog,Class,Args,false)->
  O=start(Cog,Class),
  object:activate(O),
  Class:init(O,Args);
new(Cog,Class,Args,true)->
  O=start(Cog,Class),
  cog:add(Cog,init_task,{O,Args}),
  O.

start(Cog,Class)->
  O=spawn(object,init,[Cog,Class,
           Class:init_internal()]),
  #object{class=Class,ref=O,cog=Cog}.

activate(#object{ref=O})->
        O!activate.

await_active(#object{ref=O})->
  O!{is_active,self()},
  receive active-> ok end.

init(Cog,Class,Status)->
  receive
    activate-> loop(Class,Status)
  end.

loop(Class,Status) ->
  receive
    {is_active,P}->
      S=Status,
      P!active;
    {O=#object{class=C},Field,Val,Pid}->
      S=C:set_val_internal(Status,Field,
          Val),
      Pid!{reply,O},
    {O=#object{class=C},Field,Pid} ->
      {S,Val}=C:get_val_internal(Status,
          Field),
      Pid!{reply,O,Val},
  end,
  loop(Class,S).
```

(a) The `object` module

```
init(O=#object{class=
    class_CacheTranscoder},[
    P_cacheSize])->
  set(O,cacheSize,P_cacheSize),
  O.

set(O=#object{class=
    class_CacheTranscoder=C,ref=
    Ref,cog=Cog},Var,Val)->
  Ref!{O,Var,Val,self()},
  receive
    {reply,O} -> ok
  end.

get(O=#object{class=
    class_CacheTranscoder=C,ref=
    Ref,cog=Cog},Var)->
  Ref!{O,Var,self()},
  receive
    {reply,O,Val} -> Val
  end.

-record(state,{cacheSize=null}).
init_internal()->
  #state{}.

get_val_internal(S=#state{
    cacheSize=G},cacheSize)->
  {S,G}.

set_val_internal(S,
                 cacheSize,V)->
  S#state{cacheSize=V}.
```

(b) The behavior module representing the `CacheTransoder`, with only a single field `cacheSize`

Figure 4.5.: The `object` module and a concrete class

In Figure 4.5 the `object` module and the `CacheTranscoder` class's generated callback module are shown. The `CacheTranscoder` from the VTS case study is described in detail in Section 3.2. In the shown translation, we use a down-sized version of the class with only the field `cacheSize`.

An object in the Erlang backend is created by invoking the `new` function. This calls an internal `start` function, which creates the process, and either activates and initializes the object or otherwise adds an `init_task`, if it runs on a new COG, as can be seen in Figure 4.5(a) Lines 1–8. The `start` function spawns a process, which executes the `init_task` function. The COG, the class module and the initial state are passed as parameters to this process. The initial state is created by invoking the callback `init_internal` in Line 12. This state term, which is passed along to the callback functions in all field accesses, is represented by a *record* as can be seen in Figure 4.5(b) Lines 17–19.

Records are syntactic sugar built on top of tuples and are defined like in Figure 4.5(b) Line 17. The tuples first element is an atom with the record type, followed by all other fields. A special syntax allows to access a field by its name, which is translated by the compiler to its position. A field can be matched by `#state{cacheSize=S}`, where the field `cacheSize` is matched to the variable `S`.

A new `object` process executes the `init` function, which waits on an `activate` message. This message is sent by the exported `activate` function, shown in Figure 4.5(a) Lines 15–16. Upon receiving such a message the `loop` function is executed. This function handles messages, updates the internal state and then loops tail-recursively with the new state `S`. It responds on `is_active` requests in Lines 29–31, which are used for the `await_active` function. The two other `receive` clauses over the Lines 32–37, receive field access messages and invoke the callback functions. These messages are sent by the `get` and `set` functions in Figure 4.5(b) Lines 5–15. The exported functions are generated in the `CacheTranscoder` class module, because they validate via the match in the function headers, if an object of the right class is passed. The update of the fields is performed by the `get_val_internal` and `set_val_internal` functions, by modifying the record `S`.

A complete translation of the class `CacheTranscoder` is shown in Figure 5.6 and Figure 5.7 on the Pages 53–54.

Concurrent Object Group A COG has to manage and keep track of all its tasks, which is done by a tail-recursive looping process. New tasks can be added via the functions `cog:add` and `cog:add_and_notify`, where the latter automatically sets the caller up for a notification about the task's termination. To schedule a task, the COG passes a token to the to-be-scheduled task and waits then for the return of the token. When a task reaches a schedule point or terminates, it passes the token back. Correct implementation of this behavior allows only one process in a COG to execute. With the `new_state(Cog,Task,State)` function a task can set its execution status (described in the following paragraph on tasks) to `State`.

Task A task is one scheduling unit. Different kinds of tasks are implemented by a behavior module, which requires the functions `init` and `start`. They are called by the `task` module. `init` gets the parameters, the task is started with and performs necessary initialization. For instance for an asynchronous call, the check and wait for the object's initialization status happens in this phase. After that, a task's state will be set to *Ready*, allowing it to be scheduled. At the first schedule the `start` function is invoked. If this function returns, the task state *Done* is reported back to the COG. For a description of the task states see below.

Implementation-wise we distinguish between three task types:

main_task is given a module with a main method and it invokes this method. This task is created when the runtime starts.

init_task initializes an object by calling the `<class>:init(Obj,Args)` function. This task is used to initialize an object on a newly created COG.

async_call_task represents an asynchronous call. It takes as parameters the object, the method's name and parameters. The future is also represented by this process (for further description see below). This task can further be used to implement the active object behavior, by asynchronously calling the `run` method at the end of the object's initialization.

For each task a COG stores a value describing its execution status, which influences scheduling decisions. Following states exist:

Running: task is allowed to execute by the COG, so it has currently the token

Ready: task is ready to be executed, therefore it waits on the run token

Waiting: task is actively waiting on a condition to fulfill (e.g., awaiting the termination of an asynchronous call)

Waiting_poll: task is waiting on a condition to fulfill. To check this condition the task has to be polled. For further explanation see the paragraph on the **await** statement's implementation on Page 52.

Done : task's execution is finished. Cleanup has to be done, before it is destroyed.

The `task` module provides functions for the task behavior's implementation to handle a change of its state. Furthermore, the `task` module provides functionality to await a task's termination. By calling `task:notifyEnd` a process will receive a message on the termination of this process. Afterwards the callee can block with the function `task:join` until the specified task terminates. This functionality is for example used to start a main task and await its termination.

Future A future is realized as part of an `async_call_task`. After this task is finished, it passes the token back, but does not terminate. Instead it waits for messages in a loop. To support the **get** expression and the **await** statement, messages can either ask if the computation is finished or fetch the value of the future. The latter is performed by the exported function `get(Task)`. As long as the task is still computing, it does not respond to these messages, therefore the process's mailbox is effectively used as a wait queue.

The communication and process structure in an example In the following example the realization of an ABS program, using the above described structure, is shown. A sequence diagram shows the used processes, the messages exchanged by them and the creation of processes.

A simple asynchronous call is used as an example. It depicts object creation, usage of futures, scheduling and setting of an object member variable. In Figure 4.6 the ABS model is shown. The main block, shown on the left, is the starting point of the model's execution. It creates an object of the class `Output_i`, calls the `print` method asynchronously and waits for the future to resolve.

The class `Output`, shown on the right, has a single very simple method. As ABS has no print functionality, that – as one would expect – writes to *stdout*, the shown `print` method sets just a member variable to remember the last value.

The sequence diagram in Figure 4.7 shows the execution of this model in Erlang. Arrows with filled heads represent synchronous messages (in Erlang implemented via an asynchronous message with an immediate selective receive), while unfilled arrow heads represent asynchronous messages. Parameters in messages were omitted in favor of clarity. The diagram and the corresponding behavior in the model will now be discussed from top to bottom.

At the beginning of an execution, an initial `COG` is created. This `COG` is initialized with the `MainBlock` task. After the startup of the initial components, the `MainBlock` task is scheduled by sending a *token* message to this task.

The creation of the object in Line 1 involves creating a new `Output:Object` process, which holds the object's state. As this object is created in the same COG, any code in the constructor is executed at this point by the `MainBlock` task. For this class this is only the

```
1 {
2   Output o   = new local Output_i();
3   Fut<Bool> f= o!print("Hello World");
4   await f?;
5 }
```

(a) Main block

```
1 class Output_i implements Output
2 {
3   String last= "";
4   Bool print(String s){
5     last= s;
6     return True;
7   }
8 }
```

(b) The class `Output`

Figure 4.6.: Simple asynchronous `print` call

initialization of the member `last` with the literal `""`. If an object is created on a new COG (with the **new** expression), the initialization has to be performed by a new task running on the newly created COG. In that case the `init_task` is used.

The asynchronous call in Line 2 requires to start a new process, which executes this `AsyncCall(print)` task. This is done by a synchronous message $startAsync$ to the `COG`, where the `Output:Object` resides. In our case those COGs are the same. On such a call the `COG` will start a new process. The PID, which is used as a reference to the future, is then returned to the calling `MainBlock`.

In Line 3 of the `MainBlock` an **await** on the previously returned future is executed. To be notified if the future is resolved, the `MainBlock` sends an $await$ to the `AsyncCall`, which also represents the future. After that, the `MainBlock` task returns the token to the COG and its state is changed to *Waiting*.

As the `COG` received the token, it can now schedule a different task. In this example the only other task in the state *Runnable*, is the previously created `AsyncCall` task.

The call to `print` only sets the member field `last` to the value supplied by the parameter of print. A field is set by a single message to the `Object` process. As the method returns after that, the execution part of the `AsyncCall` task is finished. In consequence the token can be returned to the scheduler and all processes, which were waiting on the future to resolve, can be notified via a message $result$.

By receiving the $result$ message, the `MainBlock` task can successfully check that its **await** condition is fulfilled. Therefore, it notifies its `COG` via a $setState$ message, that it is now in the *Runnable* state. Following from this, the *Runnable* task is scheduled by the COG. As there is no more code to execute in the `MainBlock` task, this task also finishes and thus returns its token.

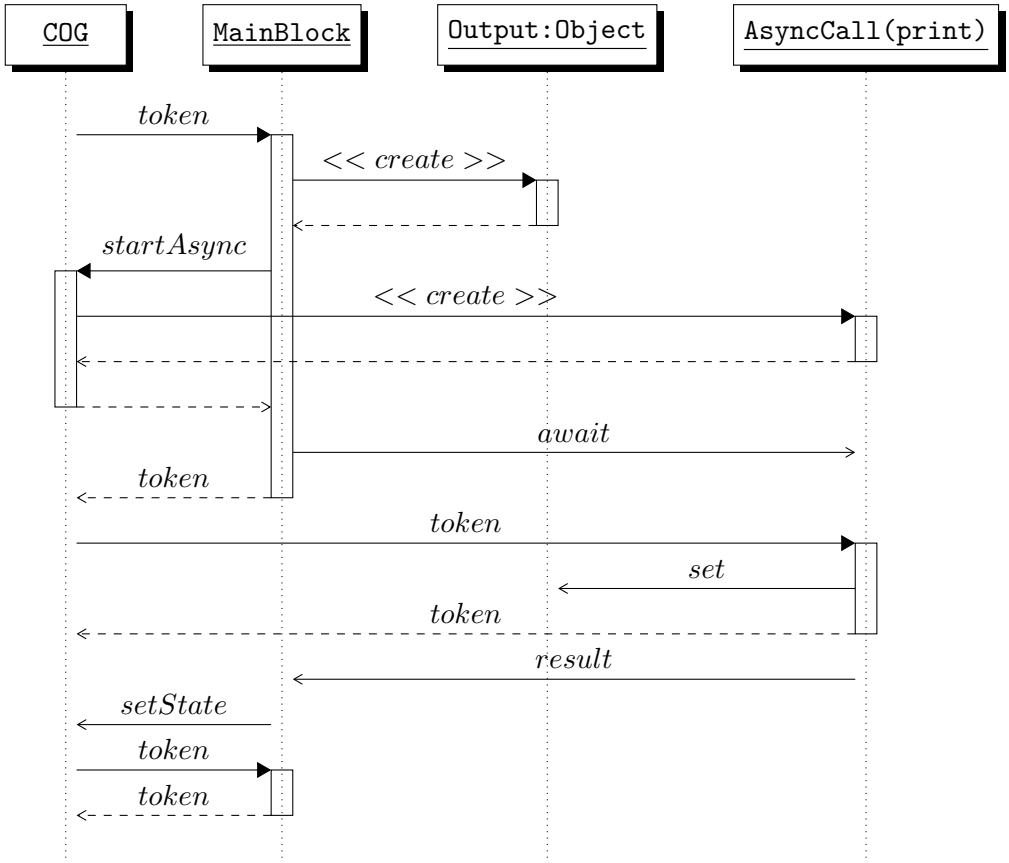


Figure 4.7.: Sequence diagram of Erlang execution of the model in Figure 4.6

Chapter 5.

The Translation Backend

To realize the concepts presented in the previous chapter, an ABS model needs to be parsed and translated to its Erlang representation. This is performed by the Erlang backend, which is covered in this chapter.

As other backends already exist, the Erlang backend will be integrated in the same existing parsing infrastructure, which will be described in the first section. In the section following, an overview of the translation process will be given and concepts, requiring a non-trivial solution, will be presented. This section closes with a side-by-side presentation of an ABS class and its Erlang translation.

Integrating the backend into the existing unit testing framework of the toolsuite increases the confidence, that the Erlang backend behaves according to the semantics. The testdriver required for this integration is explained in Section 5.3. Lastly, it is described how a translated ABS model can be executed and how this backend compares to the Java and Maude backends, which were initially shipped with the ABS toolsuite.

5.1. Existing Infrastructure

The Highly Adaptable and Trustworthy Software using Formal Models (HATS)[1] project developed a toolsuite for ABS in Java, which has tools for analysis, development support, parsing and code generation [WAM+12]. As the Erlang backend and further language additions integrate in some of those tools, an overview of their internal workings will be given.

The process of code generation consists of three steps. The first two are called the *frontend* and the last is performed by one of the *backends*:

Parsing: The textual representation of an ABS model is tokenized and transformed to an Abstract Syntax Tree (AST).

Semantic Analysis: At this step the AST is only known to be syntactically valid. Therefore, to verify the semantic correctness, checks like the following are performed: if all used types are defined and used in a compatible way; that only defined methods are called; that there is only a single declaration of an element.

Code Generation: Starting from a correctly typed AST an executable description of this model can be generated by traversing the tree.

5.1.1. Parsing

Parsing is a problem supported by a variety of tools, of which several were used in the toolsuite. The tools use an input specification and generate a parser for ABS. They are now introduced in a step by step description of the parsing of an ABS model's text.

The first step in parsing the model's text is to tokenize it, which means to split the text in parts, so that the text is transformed into a stream of tokens. A token can be an identifier, a literal, a keyword or any other special character (like a bracket or operator symbol). The definition of the tokens can be found in the *Abs.flex* file. From this definition the *JFlex* tool[2], a *lexical analyzer generator*, generates the `ABSScanner` class. This *lexer* can then be used to tokenize the model's text.

The token stream is in return used by a parser to build the AST. The AST can only be composed of elements defined by an abstract syntax in the *ABS.ast* file. The *JastAdd* tool[3] used in this work, declares itself as a *meta-compilation system*. It allows to generate a class hierarchy from an abstract syntax, supports *attributed grammars*, allows to add methods to the classes of the abstract syntax via aspect-oriented programming and integrates with parser generators.

[1] http://www.hats-project.eu
[2] http://jflex.de
[3] http://jastadd.org

```
1 abstract Exp;
2 abstract PureExp: Exp;
3 abstract EffExp : Exp;
4 GetExp : EffExp ::= PureExp;
```

Figure 5.1.: AST Elements of ABS

The JastAdd tool generates a class hierarchy representing the AST elements from the abstract syntax defined in the *ABS.ast* file. These elements can be ordered in hierarchies of types and can have elements as children themselves. In Figure 5.1 a small extract of the abstract syntax, written in JastAdd's abstract syntax language, can be seen. It shows, that the abstract element of `Exp` has two abstract subclasses. Likewise one can interpret it as a grammar, where the nonterminal `PureExp` produces the nonterminal `Exp`. Further it shows that the element `GetExp` is hierarchical below `EffExp`, so it can be used everywhere where `EffExp` can be used. In Line 4 a child is added to the `GetExp` via the `::=` expression. The child, a `PureExp`, is the expression, which represents the future, the **get** operates on.

The parser class *ABSParser* is generated by using a combination of the *JastAddParser*[4] and the *Beaver Parser Generator*[5]. The JastAddParser unifies the JastAdd tool and Beaver. The latter builds a parser from a grammar, where the class hierarchy from the JastAdd tool is used as AST elements.

The parser is built from a grammar definition in *ABS.parser*, which contains rules that match terminals and nonterminals and in case of a match produce an element of the AST. The rule for the **get** statement can be seen in Figure 5.2, which parses an expression in the form `p.`**get**, where `p` is any expression of the type future. On the left hand side of the equation in Line 1, is the name and return type of the rule. The right hand side shows the part that needs to be matched, where `pure_exp_prefix.p` is a nonterminal or a reference to the rule `pure_exp_prefix` and its result is available here under the name `p`. `DOT` and `GET` are terminals. If this rule matches, the part in the curly braces is evaluated, which will subsequently return as result of this rule `new GetExp(p)`, where `p` is a `PureExp`.

```
1 Exp eff_exp =
2     pure_exp_prefix.p DOT GET    {: return new GetExp(p); :}
```

Figure 5.2.: Grammar rule for the **get** expression

Summing up: First the classes describing the AST elements are generated. After that the `ABSScanner`, which tokenizes the input, and the `ABSParser`, which matches rules according to a grammar and produces the AST, are generated and can then be used to parse a program to an AST.

[4] https://bitbucket.org/jastadd/jastaddparser
[5] http://beaver.sourceforge.net

5.1.2. Semantic Analysis

Analysis of the model starts with an AST that complies to the defined grammar, but there are a lot of semantic criteria that need to be fulfilled to have a valid model. Some of those requirements are:

- That all referenced variables are reachable in the current scope.
- That only calls to functions or methods existing in the scope are performed.
- That no fields or methods are defined multiple times in one class.
- That classes provide the methods required in the implemented interfaces.
- That types match their usages. For example, the condition expression in a while loop must result in a value of type `Bool`.

All these checks can be implemented by traversing the AST. As the elements of the AST are auto generated, JastAdd supports aspect oriented programming to add extra functionality to the resulting classes. There are two different kinds of aspects. Imperative aspects consist of a set of fields and methods, that are appended to the generated classes. Declarative aspects allow to add synthesized and inherited attributes and equations for these. Extra analysis and tree traversing methods are added in this way.

Analysis is performed by invoking the method `typeCheck(SemanticErrorList e)` on the topmost node, which checks, if this element is clean and invokes `typeCheck` on all possible children. Errors are added to the given `SemanticErrorList`. Therefore, an error-free model is equivalent to an empty `SemanticErrorList` after `typeCheck` was called on the root element of the AST. In Figure 5.3 the part of the *TypeChecker* aspect for the `WhileStmt` is shown. One sees that `typeCheck` is invoked on its children and it is ensured that the condition is of type `Bool`.

```
1 public void WhileStmt.typeCheck(SemanticErrorList e) {
2         getCondition().typeCheck(e);
3         getCondition().assertHasType(e,getModel().getBoolType());
4         getBody().typeCheck(e);
5     }
```

Figure 5.3.: Adding `typeCheck` method to `WhileStmt`

Generation of code in a different language can be done by transforming one or a set of AST elements to valid code in the target language. Similar to the analysis, the tree needs to be traversed, so the traversal methods are also added by new aspects. Instead of accumulating errors in a list as in the analysis phase, parts of the target code are produced. This is now explained in detail in the following section.

5.2. Code Generation to Erlang

After an overview of the code generation process for Erlang is given, we will have a more detailed look on parts of the code generation, which need more attention or are of higher interest. This section closes with a side-by-side printing of an ABS class and its Erlang translation.

Overview The Erlang backend should translate a well-typed AST to a set of modules, which conform to the ideas proposed in Section 4.1. The translation application is represented by the `ErlangBackend` main class. It is a subclass of the abstract `Main` class, which subsumes the common frontend parts in all backends.

The `Main` class already provides functionality for parsing of compiler options, loading the given ABS modules and generating a type-checked AST. A subclass, like the `ErlangBackend` only has to provide processing from this AST onward. The `ErlangBackend` does so by creating first an instance of the class `ErlApp`, which represents an Erlang application as a set of modules. It allows the creation and tracking of this set, which will contain the model components. The `ErlApp` object provides for a newly created module an instance of the `ErlangCodeStream`. It supports linear writing to this new module's file and has some built-in logic to manage indentation levels.

The core part of the code generation is the traversal of the AST. Therefore, a new aspect *GenerateErlang* is added that adds tree traversal methods, which generate Erlang modules and code, to the AST elements of interest. This is an alternative to the usually used visitor pattern, when using an object oriented language to traverse an AST.

Generation is started by traversing all children of the `Model` AST element. A `Model` element is always the root of an ABS AST. It contains declarations of modules, which can involve imports, exports, an optional main block, and function, class, interface and data type definitions. As stated before, we ignore types and therefore ignore imports, exports, interfaces and data type definitions. For all other elements a method `generateErlangCode(ErlApp ea)` is defined in the aspect.

The method `generateErlangCode` generates and fills the Erlang modules for class, function and main block declarations, according to the naming scheme presented in Section 4.1. All of these are specially named functions, where the translations of the sequences of statements and expressions form these functions' bodies.

The Erlang modules, which represent the generic parts of the translation like the `cog`, `object` and `builtin` (containing all functions defined as `builtin` in the standard ABS library) and reside in the `runtime` folder in the backend implementation, have also to be part of the resulting Erlang application. Therefore, all those files and some helpers for compilation and execution, are copied to the output location by the `ErlApp` object after the translation is completed.

Statements & Expressions After the generation of a function header for a declaration of a main block, a function or a method, according to concepts presented in Section 4.1, the actual code, consisting of statements and expressions, needs to be transformed. Through the *GenerateErlang* aspect an abstract method is added to the abstract classes `Statement` and `Expression`. The subclasses are required to provide an implementation for the `generateErlangCode(ErlangCodeStream ecs,Vars vars)` method. It should write an expression's or statement's Erlang representation to the `ErlangCodeStream`.

The `Vars` object is used to implement variable tracking, which is required to resolve differences in both languages treatment of mutability and scoping of variables. These differences and the chosen solution are presented in the next paragraphs.

Single-assignment variables Erlang provides only single-assignment variables, realized through pattern matching to unbound variables. ABS instead supports multi-assignment to local variables and object fields. The latter are not in conflict with this limitation because they are stored in the object process, but for local variables a solution is needed. We need a mutable *variable environment*, which allows us to map an identifier to a value. In Erlang mutable variables for a process can be simulated either by using the process dictionary or an external storage (database or process).

The process dictionary is a process-wide mutable key value store. If used to simulate local variables, this would require a stack emulation, as in a recursive evaluation an equally named local variable could be encountered. So this adds some complexity and makes every variable access costly. Furthermore, a general advice in the Erlang community is to use the process dictionary with care, because it undermines the locality of a process's state, which is normally only defined by the current function and the contents of the process's mailbox.

The other option is an external storage, as it is used for objects. This can be done by a looping process, which represents one stack-frame, where value access is done by messages or some other stack emulation in a database. This allows a relative clear design, but would lead to lots of messages, which need to be passed around, and once again locality of function evaluation would not be guaranteed. This is an acceptable approach for objects, as they need to be accessed by multiple tasks, so some kind of reference or sharing is necessary.

Instead of implementing a mutable environment, the chosen purely functional approach is based on how one would manually reassign a value in an Erlang function: Introducing a new variable with a similar name or one that reflects the updated meaning of the variable (e.g. `NewState=State+1`). In the automatic translation, this is done by tracking all variables, and use a counter, which is appended to the variable. A reassignment then results in an increment of this counter, and therefore in the use of a new Erlang variable. Such a representation is also used in the field of compiler optimizations and is called *static single assignment form* [CFR+91].

For example the ABS code `a=a+23` would be translated to `V_a_1=V_a_0+23`, where `V_` is a prefix for all ABS variables. This requires to keep track of all local fields and their counters through different scopes and control flow influencing statements on the translation level.

Variable scope ABS uses variable scoping that is similar to Java's, where a new block introduces a new scope. Erlang instead has only a single scope, which is per function clause, where a clause is the expression (which includes sequential composition of expressions) after a function header [Eri15a]. Therefore, the generator has to ensure that a variable introduced in an inner scope is not used outside of this scope, neither as a read variable, which would violate the ABS scope, nor as a write variable, which would result in a pattern match with an already bound variable and would in consequence fail most probably. This is achieved by tracking all variables used in an inner scope with their respective counter and on a new declaration in an outer scope a new variable with an increased counter value is used. Additional attention needs to be paid, when the number of reassignment's of a variable in different branches varies. At the end of the whole control flow statement, there must exist a single variable, which holds the most recent value, independently of which branch was executed. The generator ensures this requirement by inserting variable assignments at the end of a branch if less assignments were made then in another branch. Such an assignment just copies the value from the last used variable in a branch to the variable with the maximum counter value used over all branches. An example translation with an update copy at the end of a branch can be seen in Figure 5.4.

```
Int size= 0;
if(a == null){
  size= size - 1;
} else {
  size= size + length;
  size= size + 1;
}
```

(a) Simple if

```
V_size_0=0,
case a==null of
  true ->
    V_size_1=V_size_0-1,
    V_size_2=V_size_1;%Copy
  false ->
    V_size_1=V_size_0+V_length_0,
    V_size_2=V_size_1+1
end
```

(b) Erlang variable tracking

Figure 5.4.: Variable counter tracking

Variable tracking is required to implement the solution shown in the previous paragraphs. An instance of the class `Vars` is passed around to every translation of a statement or an expression. This class stores the usage status for each ABS variable. The status is defined by a counter, which represents the number of assignments to this name, and by a Boolean flag indicating if the variable is reachable in this scope.

One or multiple new variables can be registered by invoking `Vars:nV(<Variables Names>)`. The methods `Vars:inc(String name)` and `Vars:incAll()` allow to increment one or all (see explanation of the while loop on the following page) variable's counters, in case they get reassigned. The current Erlang variable name can be retrieved by calling `Vars:get(String name)` with the variable's ABS name as parameter, which returns the variable's Erlang name in the following pattern: `V_<name>_<counter>`.

Handling branched code flows and nested scopes by the previously introduced concepts, requires additional tracking facilities. The method `Vars:pass()` returns a new instantiation of `Vars`, which copies all scope information from the called object. It can be used for a single branch. The `Vars` instances for all branches can, when the code flows converge again, be merged back in the original scope by invoking `Vars:merge(List<Vars> branches)`. This will update the original `Vars` instance's tracking status in the following way: newly introduced variables are marked as not valid in this scope, counters of valid variables are updated to their maximum value. In addition to that the `merge` method returns for each branch/passed `Vars` instance, a code segment, which updates variables in a branch to the version with the maximum counter value.

Rational numbers are a built-in data type in ABS, but no primitive data type and arithmetic operations for rationals are available in Erlang/OTP. Thus an external library[6] is used to represent them. It provides very simple implementations of the arithmetic operations on rationals. In the module `cmp` also relational operators, which convert automatically between integer and rational numbers, were added.

The case expression can match different patterns and return an expression. An example and an explanation of the **case** can be found in Section 2.2.1 on Page 14. The Erlang representation is for the most parts straightforward, as Erlang has a similar **`case`** statement. Nevertheless, there are two differences regarding patterns, that need special consideration.

First, the ABS patterns can be matched to the object's fields. As they are part of a different Erlang process's state, namely the one representing the object, they must be retrieved and stored in a local variable, before they can be matched. Therefore, the variable tracking code has to provide previously unused names for temporary variables (prefixed with `T_`), which then can be used in the pattern.

Furthermore, if a pattern contains new unused variables, they must be tracked and marked in the outer scope to be not further used. Reuse would result in a compile-time error in the Erlang code, as a variable has to be introduced in all case branches, to be used afterwards.

[6] http://github.com/sdanzan/erlang-tools/tree/master/src/rationals.erl

The while statement There is no `while` statement in Erlang, because loops are preferably implemented by using tail-recursion or list comprehensions. To implement the **while** statement, we have to follow a tail-recursive approach with an inline function, where each loop iteration represents one function invocation. Through the lack of static analysis we do not know which variables could change inside of the while loop, therefore we have to pass all variables in scope to each function invocation. To be still able to use these possible changed values after the last iteration (function invocation), they need to be stored in new variables in the scope outside the **while** statement.

In Figure 5.5 one can see a sample translation of a **while** loop to Erlang. As it is rather complex, we explain it in detail. The syntax of anonymous functions (lambda expressions) in Erlang is presented before that, to ease the understanding of the translated while loop below. An inline function starts with **fun** and is terminated with **end**. It can have multiple clauses in the form `(<Pattern>) -><Expression>`, where the `<Expression>` is evaluated if the `<Pattern>` matches.

```
1 Int unchanged=1;
2 Int mult=1;
3 Int i   =2;
4 while(i <= 10){
5   mult= mult*i
6   i   = i+1;
7 }
```

(a) Factorial

```
1  V_unchanged_0=1,
2  V_mult_0=1,
3  V_i_0=2,
4  [V_unchanged_1,V_mult_1,V_i_1] =(
5   (fun(Inner)-> fun(Param)-> Inner(Inner,Param) end
       end)
6    (fun (Self,V_unchanged_0,V_mult_0,V_i_0) ->
7      case V_i_0 <= 10 of
8        true ->
9         V_mult_1=V_mult_0*V_i_0,
10        V_i_1=V_i_0+1,
11        Self(Self,V_unchanged_0,V_mult_1,V_i_1);
12       false ->
13        [V_unchanged_0,V_mult_1,V_i_1]
14       end
15    end)
16   ) (V_unchanged_0,V_mult_0,V_i_0)
```

(b) Erlang while translation

Figure 5.5.: Factorial as While

Due to the lack of named inline functions, we have to construct a closure to have an inline function referring to itself. Therefore, in Line 5 we create a higher-order function. Its application on a function `f` returns a function which takes parameters and calls `f` with the given parameters and a reference to the function `f`. This enables the function `f` to perform self-recursive calls.

In Lines 6–15, the real loop function is passed to the previously defined higher-order function. The loop function executes, in case the condition is still true, the given statements and invokes itself afterwards recursively. In case the condition no longer holds, the recur-

sion is ended in Line 13 and the new values of the variables are returned. The starting point for this recursive invocation can be seen in Line 16, where the values before the **while** are passed to the wrapper function generated by the higher order function over Lines 5–15. The resulting possible new values of the variables returned in Line 13 are then stored in incremented variables in Line 4.

The await statement is a conditional suspension point, where a process can only resume after the **await** statement's condition evaluates to true.

At an **await** statement a process will in general pass back its token, set its status to `waiting` and wait on the condition to fulfill. As we have already mentioned in the description of process states, the waiting on this condition can either be active or passive.

Active waiting can happen as a process waits until a future is resolved. Here the process at the **await** statement, will ask the future about its status and wait for a positive response message. Due to the monotonic nature of those conditions, the reception of such a message allows the waiting process to report to the scheduler that the condition is fulfilled and the task can be scheduled again. Therefore, the scheduler does not need to take any action for an active waiting process.

Passive waiting means that the scheduler has to poll this process to determine, if its condition is currently fulfilled. So before scheduling a new process, it will invoke a check on all passive waiting processes via a message. On such a message the processes then reevaluate their **await** condition and report the result back to the COG. Non-resumeable processes will stay in the waiting state, whereas processes with a fulfilled condition will be in an intermediate state, where they wait for the scheduler's response. After the scheduling decision, such a process receives either the token or a message indicating that it has to stay in the polling mode. In this case it has to check again in the next scheduling cycle, if the condition still holds.

As conditions can also be conjunctions of conditions, they are grouped in active and passive conditions. First the process will wait until all active conditions are fulfilled, then the group of passive conditions can be checked by polling as described above.

Full Example On the following pages, we show the `CacheTranscoder` class and its Erlang translation side-by-side to put the previously shown concepts in perspective. Each ABS statement on the right-hand side is shifted to the same line, where the corresponding translation starts on the Erlang side. We have omitted a few field access methods on the first page (highlighted via inline comments) and only show the `run` method. The translation is split above two pages. The first covers the code for initialization and the object fields. The second presents the `run` method. Now we discuss the Erlang representation from top-to-bottom and how it relates to the ABS model.

```
-module(class_Server_Transcode_CacheTranscoder).
-include_lib("abs_types.hrl").
-behavior(object).
-compile(export_all).
init(O=#object{class=class_Server_Transcode_CacheTranscoder=C,
    ref=Ref,cog=Cog},[P_bs,P_e,P_d,P_cacheSize])->
    set(O,bs,P_bs),
    set(O,e,P_e),
    set(O,d,P_d),
    set(O,cacheSize,P_cacheSize),
    set(O,cache,dataNil),
    set(O,shutdown,false),
    cog:add(Cog,async_call_task,[O,m_run]),
    O.

set(O=#object{class=class_Server_Transcode_CacheTranscoder=C,
    ref=Ref,cog=Cog},Var,Val)->
    Ref!{O,Var,Val,self()},
    receive {reply,O} -> ok end.

get(O=#object{class=class_Server_Transcode_CacheTranscoder=C,
    ref=Ref,cog=Cog},Var)->
    Ref!{O,Var,self()},
    receive {reply,O,Val} -> Val end.

-record(state,{bs=null,e=null,d=null,cacheSize=null,cache=null
    ,shutdown=null}).
init_internal()->
    #state{}.

get_val_internal(S=#state{bs=G},bs)-> {S,G};
get_val_internal(S=#state{e=G},e)->   {S,G};
%%ommited
get_val_internal(S=#state{shutdown=G},shutdown)-> {S,G}.

set_val_internal(S,bs,V)-> S#state{bs=V};
set_val_internal(S,e,V)-> S#state{e=V};
%%ommitted
set_val_internal(S,shutdown,V)-> S#state{shutdown=V}.
```

```
class CacheTranscoder(BlockStream bs,
    Encoder e, Decoder d, Int cacheSize)
    implements Transcoder{


  List<Block> cache=Nil;
  Bool shutdown = False;
```

Figure 5.6.: The CacheTranscoder object

```
m_run(O=#object{class=class_Server_Transcode_CacheTranscoder=C,ref=
    Ref,cog=Cog})->
    []=((fun(Inner)->fun(Param)->Inner(Inner,Param) end end)(fun (
        Self,[])->
        case not (get(O,shutdown)) of
        false ->[];true ->task:return_token(Cog,waiting),
            task:wait_poll(Cog),
            ((fun(Inner)->fun(Param)->Inner(Inner,Param) end end)(
                fun (Self,[])->
            receive check ->
                case cmp:lt(m_ABS_StdLib_funs:f_length(get(O,cache))
                    ,get(O,cacheSize)) of
                true -> Cog!{self(),true},
                    receive wait -> Self(Self,[]);
                    token -> ok end;
                false -> Cog!{self(),false},  Self(Self,[])
            end end end))([]),
            V_tmp281503665_0 = cog:add((fun(#object{cog=T})-> T end(
                get(O,bs))),async_call_task,[get(O,bs),m_hasNext]),
            task:return_token(Cog,waiting),
            V_tmp281503665_0!{wait,self()},
            receive {ok} -> ok end,
            task:ready(Cog),
            V_next_0 = async_call_task:get(V_tmp281503665_0),
            case not (V_next_0) of
                true -> set(O,shutdown,true);
                false -> V_block_0 = cog:add((fun(#object{cog=T})->
                    T end(get(O,bs))),async_call_task,[get(O,bs),
                    m_nextBlock]),
            V_frame_0 = cog:add((fun(#object{cog=T})-> T end(get(O,d
                ))),async_call_task,[get(O,d),m_decode,V_block_0]),
            V_tmp49848285_0 = cog:add((fun(#object{cog=T})-> T end(
                get(O,e))),async_call_task,[get(O,e),m_encode,
                V_frame_0]),
            task:return_token(Cog,waiting),
            V_tmp49848285_0!{wait,self()},
            receive {ok} -> ok end,
            task:ready(Cog),
            V_store_0 = async_call_task:get(V_tmp49848285_0),
            set(O,cache,m_ABS_StdLib_funs:f_appendright(get(O,cache)
                ,V_store_0))
            end,
        Self(Self,[])  end end))
    ([]).
```

```
Unit run(){
  while(~ shutdown){



    await length(cache)<cacheSize;









    Bool next =await bs!hasNext();









    if (~next)
      shutdown=True;
    else        {
      Fut<Block> block = bs!
          nextBlock();
      Fut<Frame> frame = d!decode(
          block);
      Block store= await e!encode(
          frame);



      cache=appendright(cache,store)
          ;
    }
  }
}
```

Figure 5.7.: The run method

The `init` function in Lines 5–13 initializes the object members with the class parameters (passed as variables with the prefix `P_` or with the value specified at the field declaration in the ABS code at Lines 8 and 9. In Line 12, the asynchronous call to the `run` method is placed to start the object's active behavior.

The `set` and `get` functions over the Lines 15–21, check with the match in their function headers that the passed object reference `O` has the right `class` field. Then they send a message with the field access to the object process and await its response.

In Line 23 a record representing the internal state is defined. It contains a field for each object member field. This record is manipulated in the behavior callbacks: `init_internal`, `get_val_internal` and `set_val_internal`. The `init_internal` function returns an empty `state` record. In the `get_val_internal` function a field name is matched in the record and then returned. A field update is performed by the `set_val_internal` function, which sets the value `V` for the corresponding record field and returns it.

The `run` method translated in Lines 36–68, has no parameters in the ABS model, therefore the translated version's only parameter is the reference to the object. The **while** loop is translated as described above via a recursive anonymous function. This translation spans from Lines 37–68. We see that the function is invoked with an empty list in Line 68 and should return such in Line 37. The purpose of this list is to update all variables in the scope outside the function. In this specific case none such exist and therefore the list is empty.

The **while** loop's condition is checked in Line 38. In case it holds, the **await** statement's passive polling is executed by a self recursive function similar to a **while** loop, over the Lines 39–48. In Lines 49–54 an asynchronous call with an immediate await is executed. This statement uses a temporary variable called `tmp281503665`.

In Lines 55–66 we see how an **if** statement is translated. In the **if** branch in Line 56 the object's field `shutdown` is set. The **else** branch is bigger and includes the placement of three asynchronous calls. We wait for the last one of them over the Lines 60–64. The result `store` is then stored in the cache be invoking the `appendright` function.

5.3. Testdriver

Part of the tooling, provided by the HATS project, are a vast number of testcases for unit testing. They check syntactic and semantic compliance of the frontend and different backends.

A testdriver is the backend testing interface for the JUnit-based test framework. It has to generate the code and execute it. To check if a test resulted in the intended behavior some observations have to be made by the test framework. In case of ABS, it was decided that a model has to terminate without an error and the result of the testcase can be found in the Boolean variable `testresult`. Therefore, the testdriver has to implement a method to retrieve the value of this variable. A very simple test from the collection of tests, which check primitive data types and operations on them, is shown below:

```
1  { Int x = 10/4; Bool testresult = x == 2;  }
```

Testing integer division

The testdriver for the Erlang backend is implement in the class `ErlangTestDriver`. The code generation and compilation is straightforward, as it only requires to call the right backend method and build the resulting Erlang code using the generated *Makefile*.

Retrieving the result requires more engineering work. The execution of the main block is implemented in the runtime in a way, that a possible return value from the main block will also be the return value of the function `runtime:start`, which is used to execute a model. So we just have to change the main block to return the `testresult`. This is done via manipulating the model's AST before code generation. The necessary nodes for the **return** are just appended to the main block's statement list. The returned value can then be printed to the standard output stream via Erlang's `io:format`.

So a unit test is executed by starting an Erlang process, which starts the model with the modifications explained above. Then the JUnit part has to monitor the process's output and see if it complies to the expected result. In case a model gets stuck, e.g by a deadlock or livelock, there is a watchdog thread, that terminates the execution after a certain timeout.

5.4. Execution of a Model

To generate Erlang code from an ABS model and execute it, the backend provides different commands. They will be discussed in the first part of this section. Then a closer look will be taken on the runtime's start-up process. Furthermore, the monitoring options that the runtime provides, will be discussed in greater detail.

Commands

`generateErlang` is a *bash* script, which resides in an ABS toolsuite build under `bin/bash`. It takes a list of options and a list of ABS module filenames. The options can be used to enable or disable different checks and features or generate a specific product according to the feature models. All options are explained with the `-help` option. The script just invokes the *Java* runtime for the class `ErlangBackend` with all necessary libraries. The Erlang code alongside with a *Makefile* will be generated as presented in Section 5.2 and placed in the `gen/erl` directory. To compile the Erlang code and generate the commands to execute the model (see below) the default target of the *Makefile* should be called via `make`.

`start [--debug] <main-module>` This *bash* script launches an Erlang shell and executes the ABS model by starting at the given ABS module's main block. After the execution is finished, one stays in the launched shell, with all the necessary runtime and model code loaded. The model can be restarted via `runtime:start([<main-module>])` or inspect the model in more detail by using the Erlang process monitor or debugger. Via the `debug` option, one can enable a more verbose output that will show all task creations, scheduling, object access and more.

`run [--debug] <main-module>` Is the non-shell version of `start` script, so it will print all output to stdout and exit afterwards.

Startup process The runtime and subsequently also the execution of a model is started via the function `runtime:start`. The essential parts of the startup logic are depicted in Figure 5.8 and will now be described in detail. In Line 1 the given `MainModule` argument is converted to the chosen Erlang module naming scheme (e.g., `VideoTranscode.Tester` → `m_VideoTranscode_Tester`). The `eventstream` module used in Lines 2–7 and 12 is the monitoring facility explained in greater detail in the following section.

```
1  Module=list_to_atom("m_"++re:replace(MainModule,"[.]","_",[{return,list},
       global])),
2  eventstream:start_link(),
3  case proplists:get_value(debug,Arguments) of
4        true ->  eventstream:add_handler(console_logger,[]);
5        false -> ok
6  end,
7  eventstream:add_handler(cog_monitor,[self()]),
8  Cog=cog:start(),
9  MainTask=cog:add_and_notify(Cog,main_task,[Module,self()]),
10 RetVal=task:join(MainTask),
11 cog_monitor:waitfor(),
12 eventstream:stop(),
13 RetVal.
```

Figure 5.8.: Model startup logic in the `runtime` module

For the runtime it is necessary to know if all COGs have finished their work. This cannot be fully determined, as one does not know if a *waiting* task ever becomes executable again, as there are **await** conditions that can be dependent on system time.

The `cog_monitor`, which is added to the monitoring system in Line 7, observes events regarding the scheduling state of all COGs. After all COGs stayed idle for configurable time span (default 1 second), a call to the function `cog_monitor:waitfor()` (in Line 11) returns and therefore, the model is assumed to be terminated.

In Line 8 we start the initial COG. As we want to receive the return value of the main block (see Section 5.3), we start the `main_task` with the `add_and_notify` function. This function sends a messages to the COG and performs the following there: The COG starts the task and puts a message in the newly started task's mailbox, which allows the caller to block on the termination of this task. The waiting and the subsequent retrieval of the return value happens in Line 9, by waiting on a message of the terminated task.

Monitoring the backend To easily implement different monitoring tools, the runtime facilitates an event system with multiple listeners. The above mentioned `cog_monitor` uses that, as does a simple console logger. It can be extend to other visualizations or analysis, for instance of scheduling decisions, the current COG, the object hierarchy or profiling of runtime costs.

The module is called `eventstream` and based on OTP's `gen_event` behavior. All relevant runtime events, like the creation of entities, scheduling and asynchronous calls, are propagated through this system to the listeners.

If the runtime is started with the `debug` parameter, the console logger gets attached to the eventstream. It outputs all events to the console. In Figure 5.9 we see the output produced by running the minimal example from Figure 4.6 on Page 40. The output is structured as triplets, containing the sending subsystem, the senders identifier (the PID, e.g `<0.34.0>`) and the event. It shows a similar but more detailed textual description of the sequence diagram in Figure 4.7 on Page 41.

```
{cog,<0.34.0>,{new}}
{cog,<0.34.0>,{new_task,<0.35.0>,main_task,[m_M,<0.2.0>]}}
{cog,<0.34.0>,{state_change,<0.35.0>,waiting,runnable}}
{cog,<0.34.0>,{schedule,{task,<0.35.0>,runnable}}}
{cog,<0.34.0>,{state_change,<0.35.0>,runnable,running}}
{cog,<0.38.0>,{new}}
{object,<0.39.0>,{new,{cog,<0.38.0>,<0.37.0>},class_Output_Output_i}}
{cog,<0.38.0>,{new_task,<0.40.0>,init_task,{{object,
              class_Output_Output_i,<0.39.0>,{cog,<0.38.0>,<0.37.0>}},[]}}}
{cog,<0.38.0>,{state_change,<0.40.0>,waiting,runnable}}
{cog,<0.38.0>,{schedule,{task,<0.40.0>,runnable}}}
{cog,<0.38.0>,{state_change,<0.40.0>,runnable,running}}
{object,<0.39.0>,{set,last,[]}}
{object,<0.39.0>,{commit}}
{cog,<0.34.0>,{state_change,<0.35.0>,running,waiting}}
{cog,<0.38.0>,{new_task,<0.43.0>,async_call_task,[<0.42.0>,
              {object,class_Output_Output_i,<0.39.0>,{cog,<0.38.0>,
              <0.37.0>}},m_print,"Hello␣World"]}}
{object,<0.39.0>,{new_task,<0.43.0>}}
{cog,<0.38.0>,{state_change,<0.43.0>,waiting,runnable}}
{cog,<0.38.0>,{schedule,{task,<0.43.0>,runnable}}}
{cog,<0.38.0>,{state_change,<0.43.0>,runnable,running}}
{object,<0.39.0>,{set,last,"Hello␣World"}}
{object,<0.39.0>,{commit}}
{object,<0.39.0>,{rem_dead_task,<0.43.0>}}
{cog,<0.34.0>,{state_change,<0.35.0>,waiting,runnable}}
{cog,<0.34.0>,{schedule,{task,<0.35.0>,runnable}}}
{cog,<0.34.0>,{state_change,<0.35.0>,runnable,running}}
```

Figure 5.9.: Debug output when running the model from Figure 4.7

5.5. Comparing Backends

The ABS compiler is designed to allow different backends to generate code in different styles or languages. Before the arrival of the Erlang backend in the ABS compiler, it was shipped with two other backends, one using Maude, the other using Java. In the following paragraphs both will be introduced and their characteristics in terms of execution time, model and runtime size will be compared to the Erlang backend.

The Maude backend translates an ABS model to the Maude rewriting logic system [CDE+02], which has language support for equational logic, sorts, subsorts and allows high-performance rewriting and searches over the state space. The classes, objects and functional definitions of ABS are represented using equational logic. Simulation of a model is done by rewriting the global model state, following the rewriting rules specified in the *abs-interpreter.maude* file. In the beginning, the global model state contains only the task for the main block, but in general all objects, COGs and tasks are part of this global state term. A rewrite in this model represents the execution of a single statement in an executable task.

The Maude backend is especially suitable to analyze models in more detail, control their execution and/or extend the ABS language for the following reasons [JHS+10, WAM+12]:

- As syntax fragments get translated almost one to one to Maude, it is easy to extend the interpreter with the rules covering additional or changed syntax and semantics.
- The global model state is a single term, and can therefore be easily inspected. A single simulation run can be represented as a sequence of state terms, which can be recorded and analyzed after completion.

Due to this structure and the ease with which one can add new rewriting rules in Maude, a lot of extension to ABS, like the user-defined scheduling by Bjørk et al. [BdBJ+13] or a cost simulation for the analysis of resource management in the cloud by Johnsen et al. [JST12], were implemented in the Maude backend.

The Java backend translates an ABS model, the standard library, all data types and the necessary runtime to Java classes. The translation and execution scheme is based on the work done with JCoBox by Schäfer et al. [SP10]. Concepts similar in ABS and Java are translated to their counterparts, like classes, objects and synchronous calls. The non-overlapping parts had to be reimplemented as new classes. For example: primitive data types, as ABS's support arbitrary size, or functional expressions, like pattern matching via the **case** expression as there is no counterpart for that in Java.

A main block is directly translated to an executable main class in Java, so translated ABS models can be used in the same manner, as one would expect to use a standard Java program. Running on the Java Virtual Machine (JVM) also allows ABS models to use other existing Java libraries, which can handle network and system operations, via a foreign function interface.

The Eclipse ABS plugin, developed as part of the ABS toolsuite, uses the Java backend to support debugging and visual inspection of a model's execution. Furthermore, the backend is also used when models become larger, as it is significantly faster than the Maude backend, as one can see in the results below.

The model and platform used for evaluation In the following paragraphs we want to evaluated execution time and model size for the Erlang backend and the previously existing backends. The used model contains two different classes, providing an implementation, that calculates the n-th element of the Fibonacci series. The calculation methods of those classes are shown in Figure 5.10.

```
Int calc(Int n){
  Int retval=0;
  if (n==1 || n==2)
    retval=1;
  else {
    Fut<Int> n1=this!calc(n-1);
    Fut<Int> n2=this!calc(n-2);
    await n2? & n1?;
    Int r1=n1.get;
    Int r2=n2.get;
    retval= r1+r2;
  }
  return retval;
}
```

(a) *Simple*: An inefficient recursive implementation

```
Int calc(Int n){
  Int i=3;
  Pair<Int,Int> state = Pair(1,1);
  while(i<=n){
    state= case state {
             Pair(fst,snd) =>
               Pair(snd,fst+snd);
           };
    i=i+1;
  }
  return case state { Pair(_,snd)=>
     snd;}
}
```

(b) *Loop*: A loop-based implementation

Figure 5.10.: Test functions to calculate the n-th element of the Fibonacci series

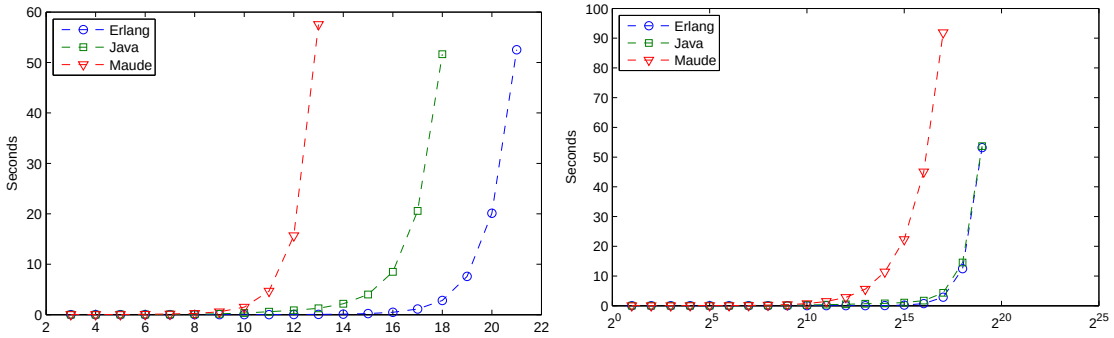
The first implementation called *Simple*, shown in Figure 5.10(a), computes recursively with asynchronous calls. As widely known, this is very inefficient as exponentially many method calls of `calc` happen. Still this a very good test of futures and asynchronous calls, as it allows the creation of very big asynchronous call trees and corresponding futures.

The implementation called *Loop* depicted in Figure 5.10(b) just performs an addition in a loop to produce the next element. It starts with the first and second elements and calculates the following element from these two. So it only needs $\mathcal{O}(n)$ potentially very

large additions and therefore is significantly more efficient then the *Simple* version. This implementation's performance is mostly influenced by arithmetical and control flow performance.

All tests were performed with the latest available development version[7] of the ABS tools from the 5th of January 2015. The Erlang backend, as presented here, does not perform garbage collection of the processes, representing the objects and COGs. As there is always a speed penalty for performing garbage collection, the version used for this benchmark contains a garbage collector, developed by Hansen [Han14]. The tests were run on an *amd64* Linux-3.18 kernel with an AMD Phenom™ II X6 1055T six-core processor with following software versions: Oracle Java 7 Update 71, Maude 2.6 and Erlang/OTP 17.4.

Execution time comparison To compare the execution time characteristics, we computed the n-th Fibonacci number and averaged over 10 runs and incremented the n until the overall runtime for 10 tries was larger than 1000 seconds. In Figure 5.11, we see the runtime for all problem sizes that have completed before this timeout. One has to note, that the increment of the problem size is chosen so that the algorithmic complexity for both computations grows exponentially.



(a) Using *Simple* (b) Using *Loop*

Figure 5.11.: Average execution time for calculating n-th Fibonacci number for Maude, Java and Erlang backend

In the result we clearly see that in both tests the Maude backend is for larger computations by far the slowest. Further we also see that the Erlang and Java backend perform quite similar for the *Loop* test, as its performance is mostly determined by arbitrary precision arithmetic.

Instead for the *Simple* algorithm, the Erlang backend outperforms the Maude and Java backend by far. If one ignores the benchmarking results for small n, where one measures mostly time spent in startup routines, we see a minimal speedup from Java to Erlang in the order of one magnitude and from Erlang to Maude in the order of two magnitudes.

[7] Git version: $bccdb82ece154767dbd2980b1d0a4f6fdca3e337$

So this backs our expectation, that Erlang's good performance regarding a high number of processes and messages would result in a better ABS execution performance. One has to note that the Erlang backend had a slightly higher CPU load on the operating system, than the other two. Erlang puts 1.5 cores under load, whereas Java uses 1.25 and Maude exactly one. This happens even though the *Simple* algorithm runs in a single COG and is therefore single threaded. This can be explained by the fact, that a few tasks, like resolving a future and its state change in the COG, run concurrent to the active task in the COG.

Model and runtime size in the backends We also want to compare how much bigger a translated model gets in the target languages. Therefore, we count the Lines of Code (LOC) for the Fibonacci model and the model of the VTS case study from Section 3.2 in ABS and each of the backends. Additional we count the size of the runtime library, that is required to execute a model in a backend. We use as definition for a LOC a physical line in a file, while omitting whitespace and comments. This cannot be seen as a measure of complexity as code can be very differently written and each language has a different, but hard to determine, expressiveness per LOC.

The results for the model, containing both Fibonacci implementation classes, are shown in Table 5.1. In the first column we see the size of the model in ABS and in the backend translations. In parenthesis the growth in percentage in comparison to the ABS source is shown. We see the Maude backend produces output that is smaller than the original source. This happens due to the fact, that type information is omitted in the information and all code of a class or function is printed into a single line. The Java output is significantly larger, as it contains a lot of code for the visual debugger and needs verbose wrappers for asynchronous calls. In the Erlang backend the size roughly triples, which seems not surprising, as we saw in the detailed description of the translation, that very few statements require a more complex multiple lines long translation.

For the runtime size we see a slightly different trend than before for the model size. The Erlang and Maude runtime have a similar LOC count, even though from experience Maude code is much more compact. This observation can be explained by the bigger feature-set supported by the Maude runtime. It supports extensions for timed models, user-defined scheduler [BdBJ+13] and modeling resources [JST12]. The Java backend has again the largest code base, as it needs classes to support built-in types and expressions.

	VTS LOC		Fibonacci LOC		Runtime LOC
ABS	185		57		
Maude	109	(-41%)	49	(-14%)	2193
Java	1411	(662%)	400	(+601%)	7192
Erlang	581	(+214%)	138	(+142%)	2185

Table 5.1.: Sizes of Models and Runtime in different backends

Chapter 6.

Error Handling

In this chapter we introduce an error handling concept for ABS. The ABS toolsuite lacked any notation of runtime error or error handling in the version available in mid 2013. The error handling introduced here and also presented in [GAJ+13, GJSS14], is inspired by concepts found in the Erlang programming language. Erlang's actor-style processes are conceptional similar to active objects with asynchronous message invocation. Moreover, Erlang is well known for its fault tolerance and successful use in high availability systems.

After we show how we define errors and why error handling is a worthwhile addition to the ABS language, the Erlang error handling capabilities, the proposed language modifications and their implementation is presented.

These new capabilities will then be used to enhance the Video Transcode Server case study. Furthermore, the error handling primitives allow the implementation of supervisors, an Erlang concept for overseeing other processes or, in case of ABS, objects.

6.1. Faults, Errors and Failures and why model them?

First, we want to define what an error is. To solidify our understanding we show a definition for each of the three terms, faults, failures and errors, and the relations between them. The definitions give also reasoning that errors are actually a concept we want to include. The terminology, used in this thesis, is based on overlapping concepts of Johnson [Joh88] and Lee et al. [LA90]. There exist a variety of slightly different definitions by others, and we compare ours with an Institute of Electrical and Electronics Engineers (IEEE) standard. In the later parts of this section, we argue why we want to have the possibility to include errors and error handling in a model.

Faults are manifestations of incorrect behavior, that in turn could lead to errors, which are defined as an erroneous state (see below). A fault is observable at some internal level, like a malfunction circuit or a faulty part of the source code. Faults reside in hardware or software and can be introduced in different phases of a system lifetime. They can be caused by specification, implementation and runtime mistakes or disturbances. If a fault is present, it does not have to lead to an error, for instance if it is located in a part of a model that is never executed.

Removing a fault can be very hard or is potentially not possible at all. Specification and implementation mistakes are introduced in the design and construction phase of a system and thus can only be removed by a redesign. It is even more difficult to eliminate faults that happen through the physical representation during the runtime of a system. Each system needs to have such a physical representation to be used. Such faults can be in the electrical components, power supply and input from the environment. Some of the sources, leading to such faults, cannot be eliminated.

Errors are defined as the occurrence of an erroneous state in a model. Such a state is internally visible in the model and can manifest itself, for example, as a wrong value or behavior caused by a fault. If an error can be detected, a model could reach in some situations again an error-free state via choosing the right transitions. Such a model is able to correct certain errors. In general, there are errors that can, depending on the capabilities of the system, not be detected.

Failures are deviations of the observed system's behavior from the expected behavior caused by an error. Errors are internally visible error states, whereas failures are observable from the outside, as the model's output or behavior. Other literature does not distinguish as clearly between a failure and an error

The relationship of faults, errors and failures has a lot of different perspectives. There is a causal relationship that a *fault* can cause an *error*, which in return can cause a *failure*. Johnson [Joh88] proposed a three universe view. A fault happens in the *physical universe*, an error in the *information universe*, as it affects the state of a system and a failure is visible in the *external universe*.

Furthermore, if one views a model as a composition of components, one can also say that a failure of a component is a fault in the model using this component. So these definitions also depend on the detail with which one looks at a system. An example should further clarify this. A distributed model runs on two machines, exchanging messages via a serial connection. If now an environment related fault happens, like somebody trips over the wire between those machines, the missing signal will result in an error. Following that, the connection cannot transmit anymore and the expected behavior of the connection will not match the observed behavior. So we have a failure in the connection. If the distributed

model only sees the connection as a component, where the correct behavior is shown by the timely transmission of messages, this kind of failure will show up as a connection fault in the model. This fault could, when the model tries to use the connection, bring the model in an erroneous state and potentially lead to a failure.

From the relationship between errors and faults, other equivalent definitions can be derived. Software systems are often attributed as *fault tolerant*. Because an error is a manifestation of a fault, these systems can also be seen as capable of *error handling* or *error recovery*.

The IEEE standard over the *Classification for Software Anomalies* [IEE10] is compared to the above introduced definitions. The IEEE standard focuses solely on software in contrast to the definitions used in this thesis, that stem from embedded systems, which have to focus on hardware as well. The failure term is defined similarly as behavior outside of defined limits. The error term is omitted and instead a fault is defined as a defect that is encountered during an execution. Defects are a supertype of faults and are in general observable causes in software system. A defect is not a fault, if it is found via other analysis tools, like static analysis or manual inspection. The IEEE's defect definition is therefore closer to our fault definition.

Including errors in modeling is interesting and valuable from multiple perspectives. First, it allows to make errors that are caused by unavoidable faults in the physical environment explicit. An analysis of failures in distributed computing and the underlying causes by Gray [Gra86], clearly shows that errors are common and one has to take them into account. He points out that especially communication failures are a common problem as there are a lot of communication paths in a distributed system.

Communication or data processing in a distributed setting are special areas, where one has to consider malfunctions in a system. Algorithms used there have to take a certain rate of communication failures and unavailability of hosts into account as shown by Lamport [Lam78] or Schneider [Sch90]. Therefore, one should also include these kinds of failures in a model, when one wants to specify a distributed software system. One could also model such potential transmission errors via timeouts and introduced delays in a custom way, but we believe that an explicit specification of errors is more concise and allows analysis targeted at the built-in error semantics.

Furthermore, it is also interesting to include error handling code in models to end up with a model, which is closer to an implementation used in production systems as those always have to include error handling.

6.2. Erlang's Error Handling

One of the requirements while designing Erlang was *Fault tolerance both to hardware failures and software errors*. Joe Armstrong had a specific idea on how to support fault tolerance [Arm07a]:

> *You can not build a fault-tolerant system if you only have one computer. The minimal configuration for a fault tolerant system has two computers. These must be configured so that both observe each other. If one of the computers crashes, then the other computer must take over whatever the first computer was doing.*
> *This means that the model for error handling is based on the idea of two computers that observe each other.*

This concept was transferred to Erlang. Instead of computers, processes need a facility to observe other processes. The primitives provided by the runtime are linking and monitoring between processes, which then enable a process to supervise other processes and automatically restart them. These concepts will be discussed in detail in the following paragraphs.

Errors An error in an Erlang process can occur at any point and is raised by the built-in functions `error`, `exit` and `throw`. These can be handled by the well-known *try catch* exception mechanism, or, in the case of an unhandled exception, lead to the process's termination, where the exception's value will be the process's exit reason.

Link and Monitor To enable mutual observation of processes, a process can link itself to another process. If one of these two processes terminates, the runtime environment sends an *EXIT* message to the other process, which contains an exit reason. Unless this exit reason is `normal` (termination because the process reached the end of the function), the linked process will terminate too and in consequence propagate its own *EXIT* message to its other linked processes. With this feature, referred as *error propagation*, it is possible to let groups of processes or even the whole system crash, which enables automatic termination and clean up of components, consisting of multiple processes.

To enable processes to observe *EXIT* messages and react on them, a process can be marked to be a system process with the *trap_exit* process flag. Such processes will not terminate when receiving an *EXIT* message, instead they can retrieve this message from their mailbox.

A *monitor* is an asymmetric form of a link, where no exit messages are propagated but special monitor messages, which indicate, if the monitored process did not exist at the point of observation or has terminated. This feature is used for example, when processes communicate synchronously in a *Request-Acknowledgment* style, to monitor if the other process stays alive, while waiting for an acknowledgment.

Supervision Tree With the *trap_exit* flag and links, it is possible to build a network of processes, with some dedicated processes observing and automatically restarting others. In Erlang this is organized as a tree. This so called supervision tree, consists of supervisor processes, which manage a set of children that can be worker processes or supervisor processes themselves [Eri15b] .

A supervisor is started with a list of child specifications and a strategy for restarting its children. The strategy defines, in case a child terminates, if only this child, all children or only children started after this child, should be terminated and restarted. Combined with the tree structure, this allows to build a variety of scenarios of shared and independent components, one normally has in a software system.

6.3. Error Handling for ABS

The presented error handling concept takes the principle of error propagation between execution units (processes) from Erlang and adds some primitives for termination of a process and a notion of error, which enables an implementation of an Erlang-style linking between objects.

In ABS an error will be propagated by the future, which is normally used to retrieve the computed value. Furthermore, an automatic rollback from the object's possibly faulty state is executed in case of an error. Reasoning for these choices and details are presented in the following paragraphs.

New language constructs are introduced to enable the envisaged error handling:

- a notion of user-defined error types
- a generalization of the future mechanism to propagate either return values or errors
- a statement **abort** `e`, which raises an error `e` and thereby terminates the process
- a statement `f.`**safeget**, which can receive both errors and values from a future `f`
- a statement **die**, which terminates the current object and all its processes

The occurrence of an error is represented in the model by means of the statement **abort** `e`, where `e` is a user defined error. These errors should be represented by a special data type. Due to the lack of extendable types in ABS, those are represented instead as a `String`. The **abort** statement is inspired by the work of Johnsen et al. [JLZ11].

Such an abort can either be explicit in the model or can be seen as implicit error in the internals of the execution. It could represent a distribution, system (e.g., out of memory) or runtime (e.g., division through zero) fault.

The semantic interpretation of such an error is dependent on the kind of ABS process that encounters it. We have defined the following behaviors:

Active Object: If the active object's process evaluates **abort**, all current asynchronous calls to this object will abort with the error `e` and the references to this object will become invalid. Further synchronous or asynchronous calls to this object will result in an `DeadObject` error on the caller side. This behavior was chosen, as the object's behavior is seen as an integral part of its correctness, and like an invalid state also an unexpected termination of this behavior leads to an inconsistent object and therefore the object cannot be further used.

Asynchronous Call: After terminating the process, an **abort** `e` statement will set the error e in the associated future. Moreover, in the callee's COG an automatic rollback (see next page) will be performed.

Main Task: As the main task represents the beginning of the execution, an **abort** there will not be further handled and the runtime system will terminate.

The automatic rollback discards all changes to the COG's objects' values since the last scheduling point, which can either be an **await** or **suspend**, when an error occurs in a task in the COG. This guarantees that objects only evolve from one well-defined state at a scheduling point to another and that, in case of an error, an object is not left in a faulty state. This is inspired by how one would handle an error in Erlang, where error recovery is performed by a restart with a previously error free state. Furthermore, it enables also to reason about the correctness of a model by using invariants over scheduling points as proposed in [DDJO12], even if the model faces runtime errors.

Extending futures to contain either the computed value or a potential error, raised either by an **abort** on the callee side or in the internals (e.g., by a distribution failure), enables error propagation over invocations. Following this, also the semantics of the `Future.`**get** statement needs do be adjusted. Inspired by Erlang's fast failing mentality and the default error propagation, a **get** will, in presence of an error `e` in the future, lead to an implicit **abort** `e` on the caller side.

The newly introduced `Future.`**safeget** allows to stop this propagation and to react on errors. The statement `Future<T>.`**safeget** returns the type `Result<T>`, which can either be of type `Value<T>(T v)`, where `v` is the result of the computation, or of type `Error(e)`, where `e` is the contained error.

The die statement allows, in asynchronously called methods, to terminate the active object they are acting upon. Its semantic meaning is the same as an **abort** in the execution context of an active object task. In other words: all active asynchronous calls and the active object's task are terminated. This behavior is in contrast to the evaluation of an **abort** in the context of an asynchronous call, where only this call would terminate and an automatic rollback would happen. This statement allows to forcefully terminate an object by an asynchronous call and is added to implement object linking.

Implementing linking Previously presented primitives enable us now to implement a Erlang-style linking between two objects, where one gets terminated in case the other terminates. The abstract implementation idea is to represent a link by two asynchronous calls, one to each of the objects. Both calls will block forever in an **await** with the condition `false`. So the future only gets resolved by an error propagated through it and thus enables the caller to perform an action in case of the termination of an object.

```
1  class Link(Linkable f,Linkable s){
2    Int done=0;
3    Unit setup(){
4     f!waitOn(this,s);
5     s!waitOn(this,f);
6     await done==2;
7    }
8    Unit done(){
9     done=done+1;
10   }
11 }
```

(a) The `Link` class

```
1  class Linkable() implements
       Linkable{
2    Unit waitOn(Link l,Linkable la){
3      Fut<Unit> fut=la!wait();
4      l!done();
5      await fut?;
6      case fut.safeget {
7       Error(e) => die e;
8      }
9    }
10   Unit wait(){
11     await false;
12   }
13 }
```

(b) A sample `Linkable` implementation

Figure 6.1.: Implementation of Links in ABS

In Figure 6.1 a sample implementation is shown, which assumes that each class implements code similar to the `Linkable` class. A link can be established by creating a new object of `Link`, where the link gets initialized with references to both objects (referred as `f` and `s`), and then calling `setup` on this new link. The `setup` method initiates the calls between the objects, by calling `waitOn` and then waits until both calls are processed. A finished call can be seen by observing the counter `done`.

The `waitOn` method implemented in the `Linkable` class, places the normally nonterminating asynchronous call in Line 3 to the other `Linkable` it should link to. The nontermination is achieved by a simple **await** `false`, as can be seen in the `wait` method. After these calls are made, the `waitOn` method reports back to the `Link` that it has succeeded. Then it will await the termination of the call in Line 5.

Should this future ever contain a value, it must be an error. In Line 7 we can perform an action, in case the other object has terminated, which will trigger in return a subsequent termination of the executing object via the **die** `e` statement. To achieve the same behavior as in Erlang, where a system process handles an exit message, one could insert custom code instead of the **die** statement in Line 7.

The setup of a link and the behavior in case of a termination of one of the `Linkables` is also depicted in the sequence diagram in Figure 6.2. It should help to get a better understanding of the interaction between the `Linkables` and the `Link`. The diagram illustrates a scenario where we already have the existing objects `a` and `b` of type `Linkable`, as shown in Figure 6.1, and an object of the type `Link`.

First, the main block calls `setup(a,b)` on the link. This method then places an asynchronous `waitOn` call to `a` and `b`. Subsequently both perform the same operation: they call `wait` on the other end of the link. This pending call is for clarity reasons depicted as a

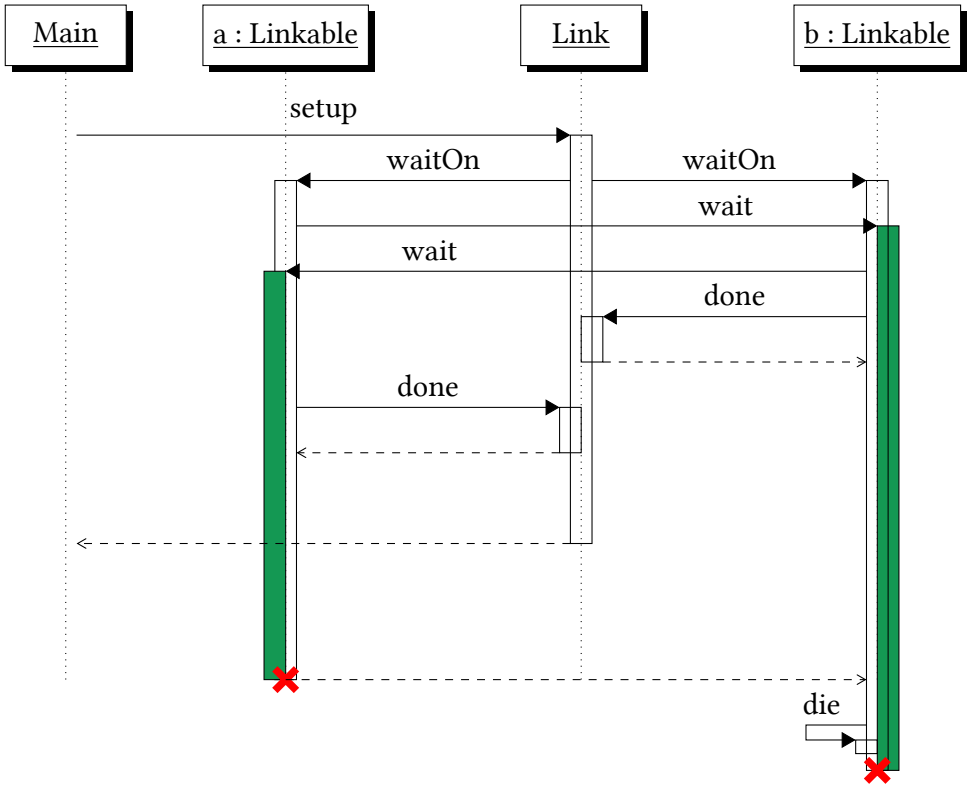


Figure 6.2.: A sequence diagram of the creation and termination of a link

green colored thread. The `wait` and `waitOn` call, which waits on the return of the former, will now run and wait till the termination of the linked object. To notify the `Link` of the completed setup, each `Linkable` is calling the `done` method. After this happened on both sides, the `setup` call returns and a link is established. The `Link` class was only necessary for the setup of the links and can now be terminated.

The interactions after the `setup` call returned, show the behavior of the termination of the object `a` and the propagation over the link. As `a` terminates, which is highlighted by a red cross, all pending tasks are terminated, therefore the green colored thread of the `wait` call returns with an error. In consequence the `waitOn` call on `b` calls **die**, which leads to its own termination.

As said before, a class needs to implement the behavior shown in the `Linkable` class. In future versions it could also be feasible to make this a default option, so that each class will be supplied with this behavior by the runtime or the compiler.

6.4. Implementation of Error Handling for ABS

Adding the above presented error semantics, requires changes in the existing translation and the addition of concepts to handle the failures. There are basically four areas, which have to be changed: extending futures to contain an error, the semantics of a failing task, object rollback, and termination of an object. In the following paragraphs we first will have a look on how errors are represented and then how those areas change.

Errors are translated to Erlang exceptions. An **abort**`("division_zero")` is translated to an `exit("division_zero")` statement. If such an exception is not handled, the process will terminate. These errors can be caught, but also observed in other processes via linking or monitoring.

Futures and Errors Previously the `async_call_task` also represented the future. As this task now can fail or be forcefully terminated by a **die**, these two different usages of the `async_call_task` process have to be separated. Thus we introduce a new process representing the future.

If now an asynchronous call is performed, first its future is created, which will in turn add and create the `async_call_task` process in the COG. To detect a shutdown of the COG while adding the task, the future obtains a monitor on the COG.

When the `async_call_task` is initialized by the COG, it links itself to the future. By setting the `trap_exit` system flag in the future, it is able to detect an error in the `async_call_task`. The future then just has to wait either on a message that the process has shutdown like `{'EXIT', _, Reason}`, where the `Reason` represents the error, or on a message with the result, that the `async_call_process` will send in case the call completes successfully. The result will be stored in the future by using a tail-recursive function, which takes the result as parameter and replies to messages from the **await** or **safeget**/**get** statements.

The error propagation of the future's **get** is implemented in the corresponding Erlang `get` function of the `future` module. There we retrieve the value of the future by sending a message and waiting for a reply. In case it contains an error value, `exit(Reason)` with the retrieved reason will be immediately called. The implementation here behaves in the same way as for an **abort**, which was also a goal defined by the error model.

Errors in Tasks The proposed semantics have differing rules for the occurrence of an error in each kind of task. In the translation, shown in Chapter 4, an object's active behavior was implemented by performing an asynchronous call to the `run` method after the initialization. As we now have to distinguish between the active behavior and other

asynchronous calls, a new `active_object_task` module was created. This process calls `object:die` in case of an error, whereas the `async_call_task` calls `task:rollback` (both are explained in the following paragraphs).

Also the COG's list of tasks has to stay in sync, when tasks face errors and terminate. This can be easily done via linking the task with the COG process and trap system messages there. So if it receives an *EXIT* message from the running process, it can remove it from the list and assume that its run token has been returned.

Object rollback To enable the automatic rollback in case of an error, both the previous and current state have to be saved until a scheduling point is reached and the previous state can be discarded. As the state covers all objects of a COG, a full copy of it could be costly, therefore an incremental checkpoint approach, as defined by Elnozahy et al. [EAWJ02], was implemented. For this we have to track firstly changes in objects and secondly which objects have changed.

The first part was implemented by storing updates to object's members in a key-value datastructure. These updates are only applied to the object's fields at a commit. As the object process was already quite complicated, it was changed to be a `gen_fsm` behavior. This is an Erlang/OTP pattern for implementing a server with an internal finite state machine. It allows to have an object in two states (uninitialized or active), and have synchronous and asynchronous method calls with simple call handlers.

To track which objects have changed in a COG, each COG got an `object_tracker` process. It is a simple `gen_server`, which is another Erlang/OTP pattern for a plain looping process with an internal state. It allows to add objects to a *dirty set*, which represents all objects with a changed state, and retrieve and clear this set.

Every time an object's field is changed, this value is stored in an *updates map*, and it is marked dirty by the `object_tracker`. If `task:rollback` is invoked, the *dirty set* of the COG is retrieved and cleared and the *updates map* gets emptied for each object in the set.

Object's die An object terminates, if an error occurs in the active object behavior or the **die** statement is applied on an object. The new semantics define that in case of a termination all incomplete asynchronous calls are aborted with the same error and the object reference becomes invalid. Every synchronous and asynchronous call to such an invalid object results in an implicit **abort**`("DeadObject")`.

To terminate each pending asynchronous call on an object, we have to track those per object. Therefore, when an `async_call_task` checks, if an object is already active, it is put in a list saved in the `object`'s internal state. By having a monitor from the `object` to the `async_call_task`, the termination of a task is easily detected and the list is kept up-to-date. In case of the object's termination, only an *EXIT* message has to be sent to each of these processes and then the object process can terminate.

Now we still have to handle the scenario where one tries to access an invalid object. Such a reference is represented by a non-existing object process. For an asynchronous call this check happens implicitly as it is checking if an object is already in an initialized state. Whereas a synchronous call normally only executes the corresponding class's code and only interacts with the object process when a field access happens. Therefore, we have to make a simple synchronous communication attempt with the object process before each synchronous call in order to check if this process is still running and otherwise execute an **abort**`("DeadObject")`.

6.5. Error Handling in the Case Study

Before error handling is introduced in the model of the case study, possible error scenarios should be discussed. In this model following errors are considered:

Read errors occur when the `BlockStream` is accessed and for instance the source file is not available anymore or cannot be read.

Decoding errors occur when a successfully read block will be decoded but somehow this block does not correspond to the specified codec.

Network errors on the client side occur when either a transmission error happens between client and server or somehow the client connection is unexpectedly closed.

The first two errors occur both in the *pipeline*, hence these will be further referred to as *pipeline errors*.

The new error-aware semantics allows to inspect the case study's behavior in previously presented error scenarios. As the pipeline elements (`BlockStream`, `Decoder` and `Encoder`) are blackboxes from the perspective of the transcoding server, we will not regard them in detail and only assume that an error there is correctly expressed by an **abort**.

In Figure 6.3 all objects used in the Video Transcode Server are depicted. It is a extended version of Figure 3.7 from Chapter 3. It illustrates how an error, which will be discussed below, spreads through the objects. Blue Arrows are pointing from the callee to a consumer of its future's value. Along this arrow, a potential error will propagate. The green arrows show a link between two objects, which are introduced on the following page.

Important in case of an error is, that either all used objects are cleanly terminated or another fault-handling action is taken, but errors must not happen unnoticed. Especially active objects like the `CacheTranscoder` and `ConnectionHandler` need to be explicitly terminated as their running active behavior does not allow to garbage collect these objects. Other nonactive objects like the `Encoder` can be automatically destroyed, after the last reference to them is gone (e.g., when the `CacheTranscoder` is terminated).

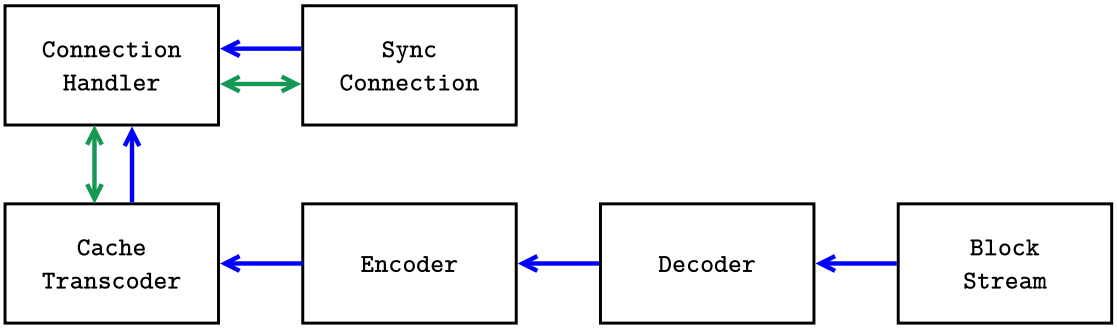


Figure 6.3.: Call structure and linking of the Video Transcode Server's Objects

Looking at *pipeline errors*, we see that an error in one of the elements will be propagated by using a **get** on the future, containing the previous result, through the end of the pipeline. So a *pipeline error* will lead to an implicit **abort** in the `CacheTranscoder`, where the resulting block is accessed. This will further result in the termination of the `CacheTranscoder`, where all pending asynchronous calls from the `ConnectionHandler` will terminate with the propagated error and all future calls will raise a `DeadObject` error. One way or another, an error will bubble up in the `ConnectionHandler`'s `run` method, where it also will terminate this object, but the connection will stay untouched and will not receive notice that the server handling part has shutdown. As we see, without changing the model the error would propagate through the server-side objects and terminate them, but leave the connection stalled.

Considering now a *network error*, which could only be observed by either sending or receiving a message on the `Connection`. Once again, we assume that such an error is contained in the future attached to those calls. An error stored there would lead to the termination of the `ConnectionHandler` (by using **get** and a following implicit **abort**). Because of the one-way communication pattern between the `ConnectionHandler` and the `CacheTranscoder`, the latter will never take notice that its consumer has terminated and therefore the `CacheTranscoder` and its *pipeline* will produce frames until the cache is full and then wait.

After looking at both error scenarios, we see that the principle of error propagation leads to a fail-fast system, but not all objects are cleanly terminated.

Introducing linking in the model allows to better resolve the parts in the error handling, which were not correctly solved before. This was due to the fact that parts of the model like the `ConnectionHandler` and the `CacheTranscoder` are mutually dependent, but this fact is not represented in the model.

A link between the `ConnectionHandler` and `CacheTranscoder` is implemented, so that both classes implement the behavior shown in the `Linkable` class in Figure 6.1 on Page 72. This link is established by adding the setup of the link in the `ConnectionHandler`, after the instantiation of the `CacheTranscoder`. This is done via **new** `Link(this,transcoder)`, where `transcoder` refers to the newly created object.

A second link is added between the `ConnectionHandler` and the `SyncConnection`. A failing link should not lead to the termination of the `Connection` object, but should call the `SyncConnection:close` method, to signal a failed connection to the client side. Therefore, we implement a changed version of the standard `Linkable` implementation, shown in Figure 6.1, in the `Connection` class. As depicted in Figure 6.4, we modified Line 7, which determines the behavior in case of a terminating partner to call `close()` instead of **die** `e`.

By adding these links we gain a model, where errors are distributed through all related objects and necessary cleanup can be performed. This can also be seen in Figure 6.3, where the arrows showing links (green) and future error propagation (blue) form a path that communicates an error from either the network or transcoding to all active objects.

```
1  class SyncConnection() implements Linkable
       {
2    Unit waitOn(Link l,Linkable la){
3      Fut<Unit> fut= la!wait();
4      l!done();
5      await fut?;
6      case fut.safeget {
7       Error(e) => close();
8      }
9   }
10   /*Rest of SyncConnections members*/
11 }
```

Figure 6.4.: Changed link behavior in `SyncConnection` class

A more tolerant error handling The previously shown model was able to handle an error correctly by informing and cleaning up all objects. Now we want to show a version of the transcoder, which employs a different error handling strategy. This new model is based on the model with links shown in the previous paragraph.

Assuming we have an unstable source for the video streams, we want to ignore some errors up to a certain error rate, where continuing the transcoding seems not desirable anymore. The error rate at the iteration n is represented by the n-th sum of an exponential decaying sequence depicted in Figure 6.5. This error rate is also called *exponential moving average* in the field of signal processing. This rate allows to detect, if a significant number of errors occurred in recent iterations. The constant values could also be chosen differently as long as they preserve the decaying form.

$$x_i = \begin{cases} 1/4 & \text{if error} \\ 0 & \text{if no error} \end{cases}$$
$$\text{error\_rate}(n) = \sum_{i=1}^{n} x_i \left(\frac{3}{4}\right)^{n-i}$$

Figure 6.5.: Exponential moving average error rate

To compute the error rate, *pipeline errors* have to be detected in the transcoder. Therefore, the computation's results are now retrieved via the new **safeget** statement, which places the result in either an `Error` or `Value` data type. Figure 6.6 depicts on the left hand side the previous code, which was used in the transcoder presented in Figure 3.5a on Page 26 to retrieve and store the result. The right hand side shows the replacement of this previous behavior, which now computes the error rate for each iteration. This rate is stored in a newly added `Integer rate` field.

```
1 Block store=result.get;
2 cache=appendright(cache,store);
```

(a) Old

```
1  Result<Block>  r =res.safeget;
2  case  r : {
3    Error(err) => {
4      rate=rate*3/4+1/4;
5      if (rate > 2/3)
6        abort "InputCorrupt";
7    }
8    Value(store)=>{
9      rate=rate*3/4;
10     cache=appendright(cache,store);
11   }
12 }
```

(b) Active error handling

Figure 6.6.: Old behavior replaced by active error handling

These changes allow now the computation of the error rate. Should this surpass a defined threshold, the transcoding process will be aborted, as can be seen in Lines 5–6.

6.6. Supervision in ABS

As linking in Erlang enables the implementation of supervision trees, we want to show this step for ABS, which is now able to support linking as well. This allows to model a statically typed supervision tree, which maintains active objects.

6.6.1. Introduction of a General Supervisor

To achieve a versatile and reusable supervisor implementation, we require some abstractions:

- configuration of the supervisor has to be done without altering or adapting the supervision code
- abstraction of how a child gets started, as different children will need different setup routines
- different restart strategies, which define how the supervisor behaves in case of a deceased child

In consideration of these requirements, we implement a class `Supervisor`, which takes a list of `SupervisibleStarter` objects. They represent specifications on how a child can be created. The interface's only method is `Unit start(Linkable s)`, which should create a new child and build a link with the given `Linkable s`, which is one side of a link. To use a `Supervisor` as a child, an implementation of the `SupervisorSupervisibleStarter` is included.

A restart strategy is passed to the supervisor as an additional parameter. The following strategies are provided:

restart one: Only the terminated child is restarted.
restart all: If a child dies, all children will be restarted.
propagate: The supervisor and all children will terminate and the error will be thereby propagated to the next supervisor or, in case it is the root node, to the runtime system.

Managing a child requires special considerations, as the supervisor has to keep track of each child, needs to detect a link failure and be able to forcefully terminate a child for the *restart all* strategy. The standard implementation of the link mechanism, shown in Figure 6.1, has on the error receiving side no indication about the source of the link error. Therefore, we build the link not directly with the supervisor, but with an intermediate object of the class `SupervisorLink` that allows to store extra information. This class is conceptionally an *association class*, but instead for an association it stores additional data

for a link. The structure of objects and their links is depicted in Figure 6.7, where a supervisor with three children is shown. Rectangles represent objects and the green connecting lines between them represent the links.

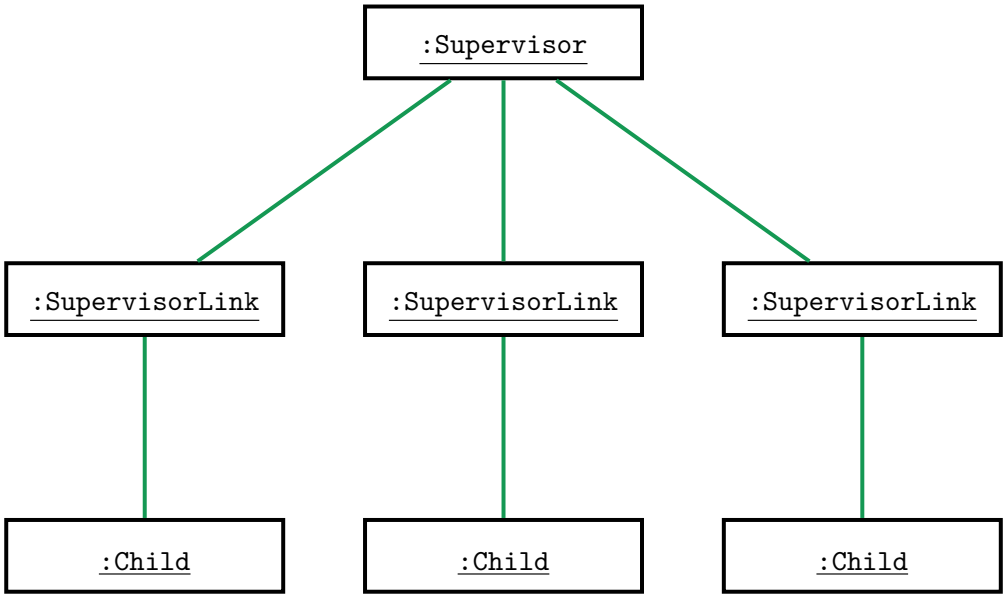


Figure 6.7.: Classes used to model a typed bidirectional synchronous connection

The `SupervisorLink` object starts the child and keeps the reference to the child specification, the `SupervisibleStarter` object, and is the other endpoint of the link, that the `SupervisibleStarter` object creates with the started child. The `SupervisorLink`'s `waitOn` method, depicted in Figure 6.8 is modified so that in case of a link error, caused by the child, it will pass the error along with the `SupervisibleStarter` object to the `Supervisor`'s `died` method.

```
1  // Object member
2  SupervisibleStarter ss;
3
4  Unit waitOn(Link l,Linkable la){
5   Fut<Unit> fut=la!wait();
6   l!done(this);
7   await fut?;
8   Result<Unit> r= fut.safeget;
9   case r {
10    Error(e) => s.died(ss,e);
11   }
12 }
```

Figure 6.8.: Adapted `waitOn` of `SupervisorLink`

This design also allows to forcefully kill a child as required for the *restart all* strategy. By terminating the child's `SupervisorLink` object, the linked child will be terminated as well.

The `Supervisor` class has to start each of the children on startup. The `start` method, shown in Figure 6.9(a), creates for each specification a `SupervisorLink`, which in turn starts the child. In this method and the illustration of the linking scheme in Figure 6.7, we also see that the `Supervisor` establishes a link with the `SupervisorLink` object. The linking allows, in case the `Supervisor` itself terminates (e. g. when the strategy is *propagate*), to terminate all `SupervisorLinks` and children as well.

When the `Supervisor`'s `died` method is called by a `SupervisorLink`, it can take action according to the defined strategy. This behavior is also shown in Figure 6.9(b). The *propagate* strategy results in the termination of the supervisor and all its children. For *restart one* the old child's specification is removed from the internal list and the child is then started again like on startup. In case the supervisor behaves following the *restart all* strategy, all current `SupervisorLinks` are terminated and then for each specification `start` is called.

```
Unit start(SupervisibleStarter child){
 SupervisorLink sl=
      new SupervisorLink(this,child);
 Link l=new Link(sl,this);
 await l!setup();
 links=Cons(sl,links);
 sl.start();
}
```

(a) Start a child

```
Unit died(SupervisibleStarter ss,
         String error){
 case strategy {
  RestartAll => this.restart();
  RestartOne => this.start(ss);
  Prop => die error;
 }
}
```

(b) Handle a deceased child

Figure 6.9.: Key methods of the `Supervisor`

6.6.2. Supervisor Tree in the Case Study

The Video Transcode Server was implemented in Section 3.2 and enhanced with error handling through linking in Section 6.5. In these models for each connection a decoder and encoder pair is started and it tries to fill the cache. Should the Video Transcode Server now face a large number of connections, each of them would start its own encoding task. This would lead to contention on the CPU, up to a level, where the tasks responsible for communication potentially run infrequently and thus make the service unusable. We will first have a look at other extensions of ABS that tackle this problem. After that we will present our solution using resource pooling and supervision.

Other approaches In the ABS modeling world the above discussed problem can be approached with two different concepts. The Envisage FP7 project and other work done by Johnsen et al. [JST15] focus on modeling of costs for different operations. The available resources in a time slice are limited for a set of COGs. Therefore, one can measure and simulate resource utilization or, in case of the Envisage project, also check if the resource usage conforms to formalized service level agreements, the model should hold. One could limit the encoding activity by assigning only limited resources to it.

Real-Time ABS by Bjørk et al. [BdBJ+13] approaches the topic of providing a guaranteed throughput by concepts of real-time scheduling. It is an extension to ABS, where one can assign deadlines to tasks running on a COG. These deadlines can either be hard or soft real-time. Furthermore, the language was extended with a construct to specify a scheduler for the tasks of a COG. Deadlines are very suitable to guarantee a certain responsiveness of an object, but as all scheduling decisions happen only in a single COG, we would end up with a design where only one encoding job can be executed concurrently.

Encoder pool The alternative approaches presented before are more usable for modeling, but are not easily implementable on fast programming language execution environments like the JVM or Erlang's BEAM (the virtual machine of the Erlang/OTP distribution). Moreover, we want to implement a solution that is closer to how it would be done in other languages.

We use the pattern of resource pooling, which globally limits the available number of a resource, in our case the encoder objects. So the pool holds a predefined number of encoders, and encoding tasks from connections are submitted to encoders of this pool.

As queuing can be easily implemented via asynchronous calls and **await** statements, the `Pool` class, shown in Figure 6.10(a) is quite simple. Its only member is a `List` of available encoders defined in Line 2. By calling `returnEncoder`, either after creation or after usage of an `Encoder`, this list is filled. The `encode` function just has to wait for an available encoder as seen in Line 7, remove it from the list, and start the encoding process.

The `Pool` and its `Encoders` are now a model-wide resource, which needs to be available for all connections. To achieve fault tolerance, they are put in a supervision hierarchy. An error in a crashing `Encoder` is propagated through the `Pool`'s `encode` method to the caller. But to circumvent the slow depletion of the pool, the `Encoders` need to be restarted. Therefore, we employ a supervision structure, depicted in Figure 6.11 around the pool and its workers, where all `Encoders` are part of a supervisor with an `RestartOne` strategy and this supervisor and the `Pool` object are supervised with a `RestartAll` strategy. This two layered structure allows a restart of the `Encoders` and the `Pool` in case of a crash. When the latter restarts, a new set of `Encoders` will be made available to the newly started `Pool` by a subsequent restart of the `RestartOne` supervisor and its `Encoders`, performed by the `RestartAll` supervisor.

```
1  class Pool(PoolReg pr) implements
       Pool,Linkable {
2    List<Encoder> available = Nil;
3    Unit run(){
4      pr!setPool(this);
5    }
6    Block encode(Fut<Frame> f){
7      await available!=Nil;
8      Encoder e=head(available);
9      available=tail(available);
10     return await e!encode(f);
11   }
12   Unit returnEncoder(Encoder e){
13     available=Cons(e,available);
14   }
```

(a) The Pool to manage the Encoders

```
1  class EncoderStarter(PoolReg pr)
       implements SupervisibleStarter{
2      Unit start(Linkable s){
3        Encoder la= new
             PoolDiffEncoder(pr);
4        Link l=new Link(la,s);
5        await l!setup();
6        Pool p=await pr!getPool();
7        p!returnEncoder(la);
8      }
9    }
```

(b) Specialized starter, which registers Encoder by the Pool

```
1  PoolReg pr= new PoolReg();
2  SupervisibleStarter ps= new PoolStarter(pr);
3  SupervisibleStarter es1=new EncoderStarter(pr);
4  SupervisibleStarter es2=new EncoderStarter(pr);
5  SupervisibleStarter es3=new EncoderStarter(pr);
6  SupervisibleStarter sup=new SupervisorSupervisibleStarter(Cons(es1,Cons(es2,
       Cons(es3,Nil))),RestartOne);
7  new Supervisor(Cons(ps,Cons(sup,Nil)),RestartAll);
```

(c) Construction of the supervisor tree

Figure 6.10.: Additions to the VTS for the Encoder Pool and its supervision structure

Two more minor obstacles still need to be resolved before one is able to use the encoder pool. First, as mentioned above, the `Encoder` has to register at the `Pool` after startup. This is done via invoking the `Pool`'s `returnEncoder` method after the start of an encoder by the `Supervisor`, as shown in the class `EncoderStarter` Lines 6–7 in Figure 6.10(b).

The second problem to overcome is the reference to the `Pool` object. Should the `Pool` crash, a new one has to be started, so a new object of the class `Pool` is created automatically by the supervisor. For all further uses until the next restart, this object should be used, thus all references to the previous object have to be updated.

Erlang faces a similar challenge as the PID changes on a process's restart. This is solved by allowing processes to register a name and provide a deeply integrated lookup mechanism for this registry. The solution in our case is heavily inspired by that. As it is not possible, due to strong typing without casts to subtypes, to implement this kind of registry in general, a simple class `PoolReg` is added. It only holds the latest reference to `Pool`, which

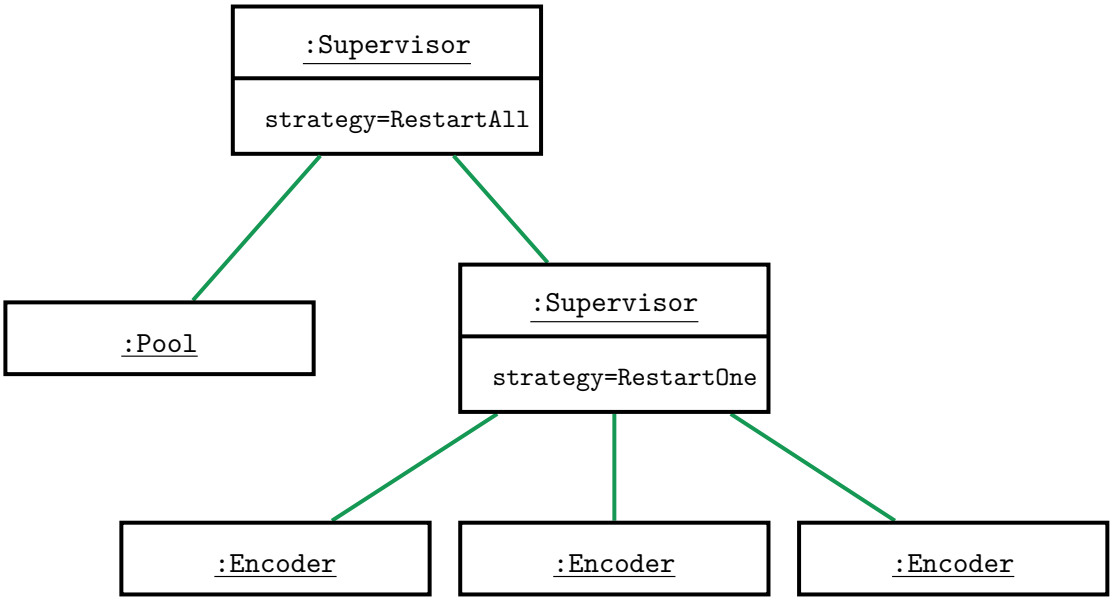


Figure 6.11.: Supervision structure for the Encoder pool of the VTS

can be accessed via `setPool(Pool p)` or `getPool()`. So all places, which need to access the `Pool`, have to retrieve the current reference from the `PoolReg`, as can be seen in the `EncoderStarter` in Figure 6.10(b) Lines 6–7.

Figure 6.10(c) shows the code that needs to be added to the main block for the initialization of the supervisor hierarchy depicted in Figure 6.11. We see how for each child a starter is specified and necessary parameters for the instantiation of an object (in our case the `PoolReg` object) are included. Furthermore, the construction of a supervisor as child by using the `SupervisorSupervisibleStarter` is shown in Line 6. To this starter we pass a list of children, that are started by the created supervisor, and its restarting strategy.

Chapter 7.

Distribution

Below we show a basic distribution approach for ABS, that allows to include distribution properties in a model and to execute it in that way. It uses an annotation to specify the location, which was introduced in a published work for simulating resource consumption. As distribution errors, occurring in this execution setting, are a main motivation for the introduced error handling, the distribution extension is built on top of the work from the previous chapter.

To understand the translation we first introduce the distributed execution in the Erlang language. Then we present a translation scheme for the Erlang backend. Finally, we look at how errors are handled in the current implementation and what further work is needed to enable a full distributed and error-aware execution of a model.

7.1. Erlang Distribution Concepts

The Erlang/OTP runtime ships with built-in distribution. Each independent Erlang runtime execution is called a *node* with a fixed identifier. This identifier, the *nodename*, can look like `testnode@testmachine.example.org` and consists of a name, chosen at startup, and the hostname. These nodes can run on the same host or a different one, reachable via a network connection [Arm07b].

Upon sending the first message to another node, these two nodes form a fully meshed network with all other known nodes via the standard distribution protocol of the runtime. To detect node failures or network errors, the nodes observe each other with heartbeat messages.

A process is normally started on the local node. Alternatively a node, where a process should execute, can be specified as an additional parameter in the `spawn` function. In the distributed runtime a process identifier contains not only a reference to the process, but also information on which node this process is running. Therefore, message passing between distributed processes works from an application perspective in the same way as it does on a single host. The message transport between the nodes is handled transparently by the distribution protocol and the runtime.

Also the error handling primitives like *link*, *EXIT messages* and *monitor*, presented in Section 6.2, work in the same manner. But if a node is network-wise unreachable and does not answer the heartbeat messages, the runtime also considers all processes on this node terminated, even though if the networking works later again as expected, and this node's processes can be contacted again. This and other pitfalls are covered in the work by Svensson and Fredlund [SF07].

7.2. Enabling Distribution in the Erlang ABS Backend

We first have to define a unit of distribution that we want to execute on different nodes. As it could be beneficial for a model to steer the placement of components, we want a representation and control of the distribution location in the model.

COGs were chosen as a unit of distribution, as they already represent a component with an asynchronous interface. Choosing a small unit like objects, does not seem advisable, as the dependence between elements of a COG could be large and therefore a lot of intra-node communication would be required. Also, as only one task is running in a COG, no benefit in terms of execution time can be expected if tasks and/or objects are distributed.

When we want to execute a COG *A* on node *N*, all task and object processes should be executed on the node *N*. This behavior is already implemented in the existing runtime, as all objects and tasks are started by the COG and therefore execute on the same node. In consequence, we only have to make sure when starting a new object on a new COG, that the latter is started on the desired node.

Deployment Components A way of encoding the execution node is still required. We chose the concept of *deployment components* introduced by Johnsen et al. [JOST11], as it represents the same idea of specifying the physical resource a COG is executed on. They used *deployment components* to model available computational resources in discrete time at a physical computation unit. To run those timed models, they used the extensions provided by Real-Time ABS [BdBJ+13].

The *deployment components* are represented by a special class/interface, depicted in Figure 7.1, which is shipped with the ABS runtime. They have a given amount of a resource, which is specified as value of type `DCData`, and methods to manipulate it. One can create a new deployment component in the same way as a new object, even if the runtime system could interpret this in a special way, e.g., starting a new cloud instance. To start a COG at a given deployment component, one has to specify it as annotation, when the COG is created.

In Figure 7.2 we see a part of the construction of the supervisor tree from Figure 6.10c. Line 1 shows the creation of a new `DeploymentComponent`, where the first parameter is its description and the second the available resources. It is set to `InfCPU`, which represents

```
1  data DCData = InfCPU
2              | CPU(Int capacity);
3  interface DeploymentComponent {
4    DCData available();
5    Rat load(Int periods);
6    DCData total();
7    Unit transfer(DeploymentComponent target, Int amount);
8    Unit decrementResources(Int amount);
9    Unit incrementResources(Int amount);
10 }
11 class DeploymentComponent (String description, DCData cpu)
12 implements DeploymentComponent
13 /* Implementation omitted */
```

Figure 7.1.: Definition of the DeploymentComponent

infinite resources, as we do not consider the amount of resources available in this model. In Line 2 we create a new object at the newly created *deployment component*. Via the build-in function **thisDC()**, one gets a reference to the `DeploymentComponent` of the current task's COG. In Line 3 of the example this is used to start another COG on the same *deployment component*, even though that is equivalent to starting a COG without an annotation.

```
1 DeploymentComponent secondNode = new DeploymentComponent("second",InfCPU);
2 [DC: secondNode] SupervisibleStarter es1 = new EncoderStarter(pr);
3 [DC: thisDC()]   SupervisibleStarter es2 = new EncoderStarter(pr);
```

Figure 7.2.: Extension of the VTS supervisor startup from Figure 6.10c: starting on different deployment components

Mapping Deployment Components to Nodes We use *deployment components* to encode the physical computation unit, but ignore the resource tracking. *Deployment components* are mapped to Erlang nodes by interpreting the `description` class parameter of the *deployment component* as Erlang nodename.

To provide easier extensibility the mapping is done by an additional Erlang process, the `nodemanager`. It is implemented as a `gen_server` behavior, which serializes the mapping operations to prevent concurrent startup of two nodes. In Figure 7.3 one sees the current logic that translates the description of a COG to a nodename. It does check if a node is reachable and otherwise tries to start this node.

The `nodemanager` parses descriptions from *deployment components* that either look like `node` or `node@hostname`. For the former it assumes that the node should reside on the localhost. After that, it tries to contact the node or start a new node by calling `check_or_start`, which is depicted in Lines 10–18. This function first attempts to ping the other node, so that, if it is not part of the meshed network, the other node will join it. Should the desired

node not be in the network after the check in Line 12, it tries to startup a new node on this host. Functionality for this is provided by Erlang/OTP's `slave` module. It basically tries to connect via SSH to the other host and start up an Erlang runtime there. This is only a very simple method of starting another node, as it requires a working public-key SSH authentication and the code of the ABS runtime and model has already to be in place. Still, it shows how dynamic startup of nodes could be integrated.

```
1  handle_call({get,Description}, _From, State) ->
2    Node=case string:tokens(Description,"@") of
3           [Nodename,Host] ->
4             check_or_start(Description,Nodename,Host);
5           [Description] ->
6             Host=net_adm:localhost(),
7             check_or_start(list_to_atom(Description++[$@|Host]),Description,
                  list_to_atom(Host));
8    end,
9    {reply,Node , State}.
10 check_or_start(Full,Nodename,Host)->
11   net_adm:ping(Full),
12   case lists:member(Full,[node()|nodes()]) of
13     true -> Full;
14     false ->
15       {ok,Node}=slave:start_link(Host,Description," -pa ebin ");
16       timer:sleep(500),
17       Node
18   end.
```

Figure 7.3.: nodemanager's logic to retrieve a reference to a node

7.3. Error Handling and Network Errors

The error handling primitives, introduced in Chapter 6, will work due to transparent distribution of the Erlang runtime in the same way. But as components of the system are now distributed, we have to take network errors into account. As the nodes are connected via TCP channels, simple network failures, like packet loss, are mitigated. So we have to consider only cases where a node loses its connection to all or a group of nodes for a longer time period. In the following paragraphs, we examine different operations' behaviors under such a network error.

A running asynchronous call is always performed at the location of the object's COG, so no network related error can occur there. Furthermore, an asynchronous call communicates with its future process. These two processes are linked together in the Erlang translation. Following a network disconnect, detected by a missing heartbeat signal, the

local runtime sends error messages for all links to the disconnected nodes. So on the asynchronous call side, this will lead to the termination of this task in the same way as an **abort** would, if the call is currently running, otherwise it will just be removed from the list of tasks of a COG. As the future receives the *EXIT* message sent from a failing link, it will take a network error as resolved value.

Placing an asynchronous call on an object that is on a disconnected node will behave in the same way as a call to a terminated object, as in both ways the object process cannot be reached. Therefore, it will result in an **abort**`(DeadObject)`. If such a network error is only temporary and the nodes get connected again, new calls to this object will work normally again.

Accessing a future at a remote node is also a special case to consider. This happens when a reference to a future, which is created at the location of the caller's COG, is passed to a COG on a different node. Communication with a future happens when using it in an **await** guard or accessing its value via **get** or **safeget**. The existing Erlang backend implementation does not assume that this communication can fail, as the future is a very simple process.

The existing semantics does not cover the case of facing a network error and not being able to communicate with the future. In the current implementation a task can fail with an **abort**, when accessing such a future. This design could lead to unexpected errors and also breaks the encapsulation of errors in futures, as there is no safe way to access a future.

Current status of the distribution We have shown that the error primitives cover network related errors already quite well. One can model and test distributed systems, where COGs can live independently on different nodes. Asynchronous calls and futures provide a way to interact with other objects in network-error-aware manner. Models have to consider that objects may be unavailable for a certain timespan only, which makes the distribution model more complex, but also reinforces the actor principle, that active objects are close to. The semantics with error handling can mostly express an error-aware distributed execution, but currently lacks the proper handling of access to remote executed futures.

Further work considering network errors is required to allow the execution of distributed and error-aware models. We have to look at futures and their location in the network and other resources that are shared over all nodes and which are potentially not reachable.

As stated above, the access to a future, if it resides on a different node, cannot be guaranteed and including errors at the level of future access would require a lot of caution and another layer of error handling by the programmer. Therefore, we envision a different design, which facilitates a node-local copy of a future.

The *eager message-based strategy*, defined by Henrio et al. [HKRZ10], could be implemented in the following way: every time a future reference is passed to a process on a different node, it has to be copied. This requires to scan all parameters, when a COG on another node is accessed. Moreover, an asynchronous call would need to be able to communicate with a set of futures, which are all linked to the call. The protocol could be formed in different ways, depending on how the semantics for futures and network errors are chosen.

If the `asynchronous_call_task` sends a message containing the value to all future copies, one cannot guarantee that all messages are received before a potential network error occurs, which would terminate all futures through linking. This behavior would violate the future's *monotonicity* property, which states that once a future has a value, it keeps it. To get rid of this undesired property, one could implement the state update of all future copies by using a two-phase commit [AHS09], which would guarantee that all futures resolve to the same value. This would require additional messages and synchronization for each asynchronous call that completes.

Another problem arises when one tries to use a globally named process that only resides on a disconnected node. At the moment only two resources are shared that way. One is the `eventstream` described in Section 5.4, which distributes events from all processes to listeners. As the listeners assume now to get all events, which is just not possible if a node is not reachable, one needs to define a new concept in case one wants to monitor nodes in that scenario as well. A solution could be node-local logs, but in general the solution is very use case dependent.

The other globally named process is the `nodemanager` introduced in this chapter, which translates the descriptions from a *deployment component* to a node and starts up a new node. The first functionality can easily be run locally, but should a net-split occur, one has to define which process is allowed to start a new node, so that one does not end up with a system containing duplicate nodes.

Due to the high complexity, open questions in the semantics and different viable approaches, above problems are not yet solved or implemented in the current Erlang distributed backend.

Chapter 8.

Related Work

Next to the Erlang backend, the ABS toolsuite ships two other backends that translate ABS models to an executable language. The *Java* and *Maude* backends are discussed and compared to the Erlang backend extensively in Section 5.5.

As ABS is an active research language, a lot of different extensions, besides the error handling introduced here, exist. The Real-Time ABS and User-defined Schedulers extensions are discussed in Section 6.6.2. A different approach on error handling for ABS's active objects is discussed below.

The author only knows of a single other work, that translates a modeling language to Erlang. Fröberg presented in [Frö93] an approach to translate the Specification and Description Language (SDL) to Erlang. SDL is a specification language, used for distributed real-time systems. In his work SDL processes and messages were translated to an Erlang equivalent. Besides using dedicated processes as message channels and timer processes, the translation is straightforward, as the concepts in both languages are similar. In consequence, his work has shown the ease and suitability of this translation, and that the resulting code is almost equal in size to the textual SDL specification. Due to the very different nature of the SDL and ABS language no proper comparison of the translation approach is possible.

8.1. Related Work on Error Handling in Concurrent Object-Oriented Systems

A lot of different approaches exist to handle errors in a concurrent object-oriented system. The concurrency aspect makes it very different from standard solutions to error handling like exceptions, as it requires the coordination of independent processing parts. We first discuss some general concepts and then approaches, that explicitly integrate error handling with futures.

Transactions and conversations Transactions are a concept initially emerged from database management systems. They allow to group a set of actions into one – as perceived from the outside – atomic action. So a transaction system performs these actions over a group of objects in a non-overlapping way. In case of an error occurring during a transaction, all performed changes are discarded by using some kind of checkpoint and rollback mechanism. Such kinds of systems perform a backward recovery, by returning to a previous state [Lis88, XRR+95].

In the conversation scheme a group of objects concurrently performs multiple actions. Correct execution can be tested in each object by some acceptance test. Only if all the tests are passed, a conversation has ended successfully. Failed acceptance tests lead to an error, which can be handled either by a rollback to a recovery point or all involved parties can perform a coordinated forward recovery to a new valid state [XRR+95].

Error-Handling in futures is approached in different ways. Futures, as introduced in Java 5 [LBG+05], handle an error on the client side, raised by an exception, in a similar way as in this work. The error is stored in the future and raised on the access by the caller. An ongoing or not yet started method call can also be stopped via a future's method.

In [JLZ11], Johnsen et al. proposed an error handling extension for active objects that is designed for ABS. The **abort** statement and the notation of the future containing the error is similar to the one of this work, but instead of error propagation and rollbacks they propose a concept of explicit compensatory actions on the caller side and additionally on the callee side. Compensation on the callee side is triggered, when the asynchronous call is canceled by the caller via using the new `f.`**kill** expression on a future `f`. Their work does not mention any special handling of the active behavior or of internal errors like network connectivity problems or resource shortage. Using compensations and explicit termination is reminiscent of conversations, as they provide means for forward error recovery.

Fabry and Noguera show an approach that only considers a volatile network connection, but not errors in general. Their approach allows to resume computation by tagging a future with a default value that is delivered, while a node is disconnected [FN08]. Should

the future resolve, in case of a reconnect, interested parties can be notified. In case of a disconnect after the future is already resolved, they integrated a mechanism that allows to reset the future's value to the tagged default.

ABS's operational semantics, which can be found in [JHS+10], is extended by an operational semantics for the new error handling primitives, presented in this work, in the work by Göri et al. in [GJSS14].

8.2. Related Work on Distributed Models

Johnsen et al. categorize modeling languages in the following way [JHS+10]:

Design-oriented languages use different kinds of representation to model the structure and the interactions and dynamics of a model.

Foundational languages are formal specifications, which use precise semantics to model certain concepts.

Implementation-oriented languages are closest to a real implementation and sometimes work as additional markup in existing programs to establish invariants.

According to Johnson et al. [JHS+10], ABS is a foundational and implementation-oriented language, as it has a formal defined operational semantics, but a model is also specified similar as in an object-oriented implementation language and can easily be translated to an implementation language as shown in this work.

A design-oriented approach that also allows to simulate resource usage in a distributed model is provided by Menascé and Gooma [MG00]. On top of a UML model one can add performance annotations, that allow to describe the used CPU time and bandwidth by a message. The model also includes explicit communication ports of a component and network resources between them. By parameterizing the network's latency and bandwidth, one can run different simulations that compute the request's response time and the number of requests per second.

The work on ABS-NET by Palmskog et al. [PDLJ13] adds a distribution model to the ABS language. In their work they focus on the communication structure and analysis of the performance and object mobility. They make nodes and arcs, which represent a network link between the nodes, a part of their model. Interaction between objects on different nodes happens transparently, by using a routing mechanism for invocations over the nodes. Their work is more focused on modeling the network and finding a good structure and communication mechanism, but does not, in contrast to this work, consider network or node failures.

Chapter 9.

Concluding Remarks

9.1. Summary

In this work, we introduced a translation to Erlang and error handling for the ABS language. Error handling is a necessary functionality for the modeling of distributed systems with unavoidable faults, that can occur in such systems. The Erlang translation formed the basis that allowed to include the new error primitives and it could be further utilized to implement distribution with support of the Erlang runtime system's distribution features. Therefore, this work started off with the Erlang translation.

First, a high level concept had to be defined for the translation. As Erlang processes resemble the Actor model and ABS's active objects are also close to this model, objects and their executing task were chosen to be represented by an Erlang process. The Concurrent Object Groups (COGs) work essentially as scheduler for all tasks in a group, so they have to be represented as independent components (processes) as well.

The translation to Erlang was integrated in the existing compiler infrastructure, which is part of the ABS toolsuite. From a type-checked AST we generate an Erlang representation. In this process, we had to find solutions for the following problems: multi-assignment vs. single-assignment variables; variable scoping; rational numbers as built-in type; await statements. The first two are handled in the compile step via tracking of all used variables. The others are handled in the runtime part of the Erlang backend, which provides utility functions and implementations for generic behavior like objects, COGs and tasks. All steps of the translation process are shown in a case study, the Video Transcode Server.

Translation is only one part of the Erlang backend. It also integrates in the unit-testing infrastructure to raise confidence in the semantic compliance of the backend. Furthermore, it provides a monitoring infrastructure and utility scripts for compilation and startup.

The error handling capabilities are heavily inspired by Erlang and support the following principles: fail fast; observe and handle errors non-locally (outside of the object/task it is occurring); restart from a previous state. This is integrated in ABS by: termination in

case of an error; storing errors in futures; default error propagation via the **get** statement; error handling via a new **safeget** statement; an implicit rollback to the state before the failing task was scheduled.

The added **die** statement, which forcefully terminates an object, allows, along with the other error handling primitives, to implement linking between objects and, in consequence, to implement Erlang's supervision trees in ABS.

The error handling capabilities were introduced to the Video Transcode Server case study. We looked at how the error semantics behaves in an unchanged model and then adapted it via linking. Furthermore, we introduced a different error handling strategy and supervision to parts of the case study.

To get closer to the goal of a distributed model with error handling, we showed an approach to modeling distribution via *deployment components* and executing such a model distributed on the Erlang backend. The error model was then examined on how it would capture distribution errors in the form of a network connection loss.

9.2. Conclusion

The main contribution of this work is an error model for active objects. The set of introduced primitives adhere to the Erlang principles of default error propagation and fast failing systems (also known as *let it crash* mentality). This relation is further manifested, as the primitives allow to implement linking and supervision for active objects, which is also shown in this work.

The error handling was evaluated by introducing it in the case study that is used as running example in this work. When introducing the error handling semantics, some error scenarios are already handled correctly in an unchanged model of the case study. To handle the case, where an object `A` stays alive, even though a terminated object `B` is the only referrer to and user of `A`, we introduced linking between these pairs of objects. With linking in place all considered error scenarios are handled by the new error semantics.

The Erlang backend provides a translation, that allows not only to execute models with the introduced error handling, but also provides a faster execution of certain kinds of models. Furthermore, the Erlang backend is now included in the official ABS toolsuite and seems to be a viable basis for additional contributions. On top of this work, Hansen [Han14] built a garbage collector for the object processes, which was a missing element of this work. Additionally, the members of the Envisage project work currently on an implementation of Real-Time ABS as a discrete time simulation for the Erlang backend.

The suitability of the error handling capabilities for distributed models is another property we have discussed. The distributed modeling and the distributed execution of such models was included in the backend. We discussed the occurrence of network errors for

such a system. In general, we are able to represent distribution errors and can express routines to handle such errors. In the error scenario, where futures are used distributed and therefore, represent non-local storage, the current semantics and the error model are not well-defined and leave open how to represent the inaccessibility of such futures.

9.3. Future Work

The introduced error model has only been used in the small VTS case study thus far. To better judge its suitability and applicability, it would be interesting to integrate it in a larger case study and compare it with other approaches to error handling.

Furthermore, fault injection seems to be a worthwhile addition to test models with error handling. When applying fault injection, faults are introduced with a certain rate at fixed points. Fault injection is also used in large scale deployed cloud applications like Netflix's *Chaosmonkey* [Net13], which kills random cluster nodes in low-load situations, while monitoring the system's health. An equal approach could be chosen for ABS.

The fault injection could also be combined with current work in the Envisage project, where they check if models conform to their Service Level Agreements (SLAs). One could test if a model conforms to an SLA under a certain error rate.

The Erlang virtual machine comes with a set of tracing features, which allow to analyze at runtime: function invocations and returns, process creation, receiving and sending of messages and other events. With match specifications one can define an Erlang pattern to select events that should be reported back. Implementing visualization or further analysis would be an interesting addition to the backend. For instance, visualizing all alive objects can be done by observing invocations of object creation functions and process terminations.

As stated in Section 7.3, the semantics for futures are not well-defined in case of a distribution error, therefore it would be interesting to define and implement such. This would then allow to implement distributed algorithms and analyze them in an error scenario, especially with fault injection in place.

Bibliography

[ABG+12] Elvira Albert, Richard Bubel, Samir Genaim, Elena Giachino, Miguel Gómez-Zamalloa, Stijn de Gouw, Reiner Hähnle, Karl Meinke, Germán Puebla, and Peter Y.H. Wong, *Deliverable D2.7 Analysis Final Report*, Tech. report, FP7-231620 Highly Adaptable and Trustworthy Software using Formal Models, HATS Project, December 2012.

[AHS09] Yousef J. Al-Houmaily and George Samaras, *Two-phase commit*, Encyclopedia of Database Systems (Ling Liu and M. Tamer Özsu, eds.), Springer, 2009, pp. 3204–3209 (English).

[Arm03] Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, Ph.D. thesis, The Royal Institute of Technology, 2003.

[Arm07a] Joe Armstrong, *A history of Erlang*, Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007 (Barbara G. Ryder and Brent Hailpern, eds.), ACM, 2007, pp. 1–26.

[Arm07b] Joe Armstrong, *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf, 2007.

[Arm10] Joe Armstrong, *Erlang*, Communications of the ACM **53** (2010), no. 9, 68–75.

[AS88] William C. Athas and Charles L. Seitz, *Multicomputers: Message-passing concurrent computers*, Computer **21** (1988), no. 8, 9–24.

[AZ98] Gul A. Agha and Reza Ziaei, *Security and Fault-Tolerance in Distributed Systems: An Actor-Based Approach*, Computer Security, Dependability and Assurance: From Needs to Solutions, 1998. Proceedings, 1998, pp. 72–88.

[BdBJ+13] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa, *User-defined Schedulers for Real-Time Concurrent Objects*, Innovations in Systems and Software Engineering **9** (2013), no. 1, 29–43.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada, *Maude: specification and programming in rewriting logic*, Theoretical Computer Science **285** (2002), no. 2, 187–243.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM Transactions on Programming Languages and Systems **13** (1991), no. 4, 451–490.

[CMP+10] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte, *Variability modelling in the ABS language*, Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers (Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, eds.), Lecture Notes in Computer Science, vol. 6957, Springer, 2010, pp. 204–224.

[dBCJ07] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen, *A Complete Guide to the Future*, Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Rocco De Nicola, ed.), Lecture Notes in Computer Science, vol. 4421, Springer, 2007, pp. 316–330.

[DDJO12] Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe, *Observable behavior of distributed systems: Component reasoning for concurrent objects*, Journal of Logic and Algebraic Programming **81** (2012), no. 3, 227–256.

[EAWJ02] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, *A survey of rollback-recovery protocols in message-passing systems*, ACM Computing Surveys **34** (2002), no. 3, 375–408.

[Eri15a] Ericsson AB, *Erlang/OTP System Documentation 6.4*, http://www.erlang.org/doc/pdf/otp-system-documentation.pdf (visited May 2, 2015), March 2015.

[Eri15b] Ericsson AB, *OTP Design Principles User's Guide 6.4*, http://www.erlang.org/doc/design_principles/users_guide.html (visited May 2, 2015), March 2015.

[FN08] Johan Fabry and Carlos Noguera, *Abstracting connection volatility through tagged futures*, Developing Ambient Intelligence, Springer, 2008, pp. 2–12.

[Fre01] Lars-Åke Fredlund, *A framework for Reasoning about Erlang code*, Ph.D. thesis, Royal Institute of Technology, 2001.

[Frö93] Magnus W Fröberg, *Automatic code generation from SDL to a declarative programming language*, SDL'93 – Using Objects, Proceedings of the sixth SDL Forum (O. Faergemand and A. Sarma, eds.), North-Holland, 1993.

[GAJ+13] Georg Göri, Bernhard K. Aichernig, Einar Broch Johnsen, Rudolf Schlatte, and Volker Stolz, *Extending abstract behavioral specifications with Erlang-style error handling*, Proceedings of the 25th Nordic Workshop on Programming Theory, Tallinn, Estonia (Tarmo Uustalu and J uri Vain, eds.), Institute of Cybernetics, 2013, Extendend Abstract, pp. 34–36.

[GJSS14] Georg Göri, Einar Broch Johnsen, Rudolf Schlatte, and Volker Stolz, *Erlang-Style Error Recovery for Concurrent Objects with Cooperative Scheduling*, Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications (Tiziana Margaria and Bernhard Steffen, eds.), Lecture Notes in Computer Science, vol. 8803, Springer, 2014, pp. 5–21.

[GMY+13] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou, *Failure Recovery: When the Cure Is Worse Than the Disease*, 14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013 (Petros Maniatis, ed.), USENIX Association, 2013.

[Gra86] Jim Gray, *Why Do Computers Stop and What Can Be Done About It?*, Fifth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1986, Los Angeles, California, USA, January 13-15, 1986, Proceedings, IEEE Computer Society, 1986, pp. 3–12.

[Häh12] Reiner Hähnle, *The Abstract Behavioral Specification Language: A Tutorial Introduction*, Formal Methods for Components and Objects - 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures (Elena Giachino, Reiner Hähnle, Frank S. de Boer, and Marcello M. Bonsangue, eds.), Lecture Notes in Computer Science, vol. 7866, Springer, 2012, pp. 1–37.

[Han14] Sigmund Hansen, *Implementing Garbage Collection for Active Objects on Top of Erlang*, Master's thesis, University of Oslo, 2014.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger, *A Universal Modular ACTOR Formalism for Artificial Intelligence*, Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, August 1973 (Nils J. Nilsson, ed.), William Kaufmann, 1973, pp. 235–245.

[HKRZ10] Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo, and Eugenio Zimeo, *First Class Futures: Specification and Implementation of Update Strategies*, Euro-Par 2010 Parallel Processing Workshops - HeteroPar, HPCC, HiBB, CoreGrid, UCHPC, HPCF, PROPER, CCPI, VHPC, Ischia, Italy, August 31-September 3, 2010, Revised Selected Papers (Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannataro, Marco Danelutto, Anders

Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, eds.), Lecture Notes in Computer Science, vol. 6586, Springer, 2010, pp. 295–303.

[HMW12] Michiel Helvensteijn, Radu Muschevici, and Peter Y. H. Wong, *Delta modeling in practice: a Fredhopper case study*, Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings (Ulrich W. Eisenecker, Sven Apel, and Stefania Gnesi, eds.), ACM, 2012, pp. 139–148.

[IEE10] IEEE, *IEEE Standard Classification for Software Anomalies IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, Institute of Electrical and Electronics Engineers, Inc., Jan 2010.

[JHS+10] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen, *ABS: A Core Language for Abstract Behavioral Specification*, Formal Methods for Components and Objects - 9th International Symposium, 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers (Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, eds.), Lecture Notes in Computer Science, vol. 6957, Springer, 2010, pp. 142–164.

[JLZ11] Einar Broch Johnsen, Ivan Lanese, and Gianluigi Zavattaro, *Fault in the Future*, Coordination Models and Languages - 13th International Conference, 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings (Wolfgang De Meuter and Gruia-Catalin Roman, eds.), Lecture Notes in Computer Science, vol. 6721, Springer, 2011, pp. 1–15.

[JOA03] Einar Broch Johnsen, Olaf Owe, and Marte Arnestad, *Combining Active and Reactive Behavior in Concurrent Objects*, Proceedings of the Norwegian Informatics Conference (NIK'03) (Dag Langmyhr, ed.), Tapir Academic Publisher, November 2003, pp. 193–204.

[Joh88] Barry W. Johnson, *Design & Analysis of Fault Tolerant Digital Systems*, Addison-Wesley Longman, 1988.

[JOST11] Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa, *Validating timed models of deployment components with parametric concurrency*, Procedings International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10) (B. Beckert and C. Marché, eds.), Lecture Notes in Computer Science, vol. 6528, Springer, 2011, pp. 46–60.

[JST12] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa, *Modeling resource-aware virtualized applications for the cloud in real-time ABS*, Formal Methods and Software Engineering - 14th International Conference on For-

mal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings (Toshiaki Aoki and Kenji Taguchi, eds.), Lecture Notes in Computer Science, vol. 7635, Springer, 2012, pp. 71–86.

[JST15] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa, *Integrating deployment architectures and resource consumption in timed object-oriented models*, Journal of Logical and Algebraic Methods in Programming **84** (2015), no. 1, 67–91.

[LA90] Peter A. Lee and Tom Anderson, *Fault tolerance: Principles and practice*, 2nd ed., Springer, 1990.

[Lam78] Leslie Lamport, *The implementation of reliable distributed multiprocess systems*, Computer Networks **2** (1978), no. 2, 95–114.

[LBG+05] Doug Lea, Joseph Bowbeer, Brian Goetz, David Holmes, and Tim Peierls, *Java specification request (JSR) 166: Concurrency utilities*, https://jcp.org/en/jsr/detail?id=166 (visited May 2, 2015), 2005.

[Lis88] Barbara Liskov, *Distributed programming in Argus*, Communications of the ACM **31** (1988), no. 3, 300–312.

[MG00] Daniel A. Menascé and Hassan Gomaa, *A Method for Design and Performance Modeling of Client/Server Systems*, IEEE Transactions on Software Engineering **26** (2000), no. 11, 1066–1085.

[Net13] Netflix Inc, *Chaosmonkey*, https://github.com/Netflix/SimianArmy (visited May 2, 2015), 2013.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden, *Software Product Line Engineering - Foundations, Principles, and Techniques*, Springer, 2005.

[PDLJ13] Karl Palmskog, Mads Dam, Andreas Lundblad, and Ali Jafari, *ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects*, Proceedings 6th Interaction and Concurrency Experience, ICE 2013, Florence, Italy, 6th June 2013. (Marco Carbone, Ivan Lanese, Alberto Lluch-Lafuente, and Ana Sokolova, eds.), EPTCS, vol. 131, 2013, pp. 85–100.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella, *Delta-oriented programming of software product lines*, Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings (Jan Bosch and Jaejoon Lee, eds.), Lecture Notes in Computer Science, vol. 6287, Springer, 2010, pp. 77–91.

[Sch90] Fred B. Schneider, *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial*, ACM Computing Surveys **22** (1990), no. 4, 299–319.

[SF07] Hans Svensson and Lars-Åke Fredlund, *A more accurate semantics for distributed erlang*, Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang, Freiburg, Germany, October 5, 2007 (Simon J. Thompson and Lars-Åke Fredlund, eds.), ACM, 2007, pp. 43–54.

[SP10] Jan Schäfer and Arnd Poetzsch-Heffter, *JCoBox: Generalizing Active Objects to Concurrent Components*, ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Theo D'Hondt, ed.), Lecture Notes in Computer Science, vol. 6183, Springer, 2010, pp. 275–299.

[Vin07] Steve Vinoski, *Reliability with Erlang*, IEEE Internet Computing **11** (2007), no. 6, 79–81.

[WAM+12] Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte, *The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems*, International Journal on Software Tools for Technology Transfer **14** (2012), no. 5, 567–588.

[Wha12] Whatsapp Inc, *1 million is so 2011*, http://blog.whatsapp.com/196/1-million-is-so-2011 (visited May 2, 2015), 2012, Blog post.

[XRR+95] Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecília M. F. Rubira, Robert J. Stroud, and Zhixue Wu, *Fault Tolerance in Concurrent Object-Oriented Software Through Coordinated Error Recovery*, Digest of Papers: FTCS-25, The Twenty-Fifth International Symposium on Fault-Tolerant Computing, Pasadena, California, USA, June 27-30, 1995, IEEE Computer Society, 1995, pp. 499–508.

[YLZ+14] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, *Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems*, Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, USENIX Association, 2014, pp. 249–265.

Appendix

Appendix A.

The Video Transcode Server ABS model

Below we show the full model of the Video Transcode Server, introduced in Chapter 3, without any of the additions introduced later in this thesis. The added error handling is depicted throughout Chapter 6.

The main module M.abs

```
module M;
import * from Server.Connection;
import Block from Server.IO;
{
  Acceptor a= new Server();

  ClientConnection c= await a!acceptNew();
  await c!clientSend("movie.mpeg");
  await c!clientRetrieve();

  ClientConnection c1= await a!acceptNew();
  await c1!clientSend("movie.mpeg");
  await c1!clientRetrieve();
}
```

The Connection module

```
module Server.Connection;
export Block;
export ClientConnection;
export Acceptor;
export Server;
import * from Server.Transcode;
import * from Server.IO;

interface ClientConnection{

Unit clientSend(String s);

Block clientRetrieve();
}

interface ServerConnection{
```

```
17
18 Unit serverSend(Block s);
19
20 String serverRetrieve();
21 }
22
23 interface ServerClientConnection extends ServerConnection,ClientConnection{
24 }
25
26 interface Acceptor{
27
28 ClientConnection acceptNew();
29 }
30
31 class Server implements Acceptor{
32
33   ClientConnection acceptNew(){
34     ServerClientConnection c= new SyncConnection();
35     new ConnectionHandler(c);
36     return c;
37   }
38 }
39
40 class ConnectionHandler(ServerConnection con){
41
42   Unit run(){
43     String fileName = await con!serverRetrieve();
44     BlockStream bs= new SimpleBlockStream();
45     await bs!open(fileName);
46     Encoder e= new DiffEncoder();
47     Decoder d=new RawDecoder();
48     Transcoder t= new CacheTranscoder(bs,e,d,2);
49     Bool closed=False;
50     while(~ closed){
51       Maybe<Block> b= await t!nextBlock();
52       case b  {
53         Just(block) => {
54             con!serverSend(block);
55           }
56         Nothing =>
57           closed=True;
58       }
59     }
60   }
61 }
62
63 class SyncConnection implements ServerClientConnection{
64
65   Int client_counter=0;
66   Int server_counter=0;
67   Maybe<String> bufferC=Nothing;
68   Maybe<Block>  bufferS=Nothing;
69
```

```
  Unit clientSend(String s){
    await bufferC==Nothing;
    bufferC=Just(s);
    Int oldVal = client_counter;
    await client_counter==1+oldVal;
  }

  Block clientRetrieve(){
    await isJust(bufferS);
    Block loc=fromJust(bufferS);
    server_counter=server_counter+1;
    bufferS=Nothing;
    return loc;
  }

  Unit serverSend(Block s){
    await bufferS==Nothing;
    bufferS=Just(s);
    Int oldVal = server_counter;
    await server_counter==1+oldVal;
  }

  String serverRetrieve(){
    await isJust(bufferC);
    String loc=fromJust(bufferC);
    client_counter=client_counter+1;
    bufferC=Nothing;
    return loc;
  }
}
```

The IO module

```
module Server.IO;
export Block;
export BlockStream;
export SimpleBlockStream;

data Block= Block(List<Int>);

interface BlockStream{

Unit open(String name);
Block nextBlock();
Bool hasNext();

}

class SimpleBlockStream() implements BlockStream{
  List<Block> source= Nil;
  Bool open=False;
  Unit open(String name){
    case name  {
```

```
      "movie.mpeg" => {
        List<Int> l=Cons(2,Cons(4,Cons(8,Cons(6,Nil))));
        source=Cons(Block(l),Cons(Block(l),Nil));
        open=True;
      }
      _ => skip;
    }
  }

  Bool hasNext(){
    await open;
    return ~isEmpty(source);
  }

  Block nextBlock(){
    Block ret=head(source);
    source=tail(source);
    return ret;
  }
}
```

The Transcode module

```
module Server.Transcode;
export *;
import * from Server.IO;

data Frame= Frame(List<Int>);

interface Decoder{

Frame decode(Fut<Block> block);
}

interface Encoder{

Block encode(Fut<Frame> frame);
}


interface Transcoder{

Maybe<Block> nextBlock();
}

class CacheTranscoder(BlockStream bs, Encoder e, Decoder d, Int cacheSize)
    implements Transcoder{
  List<Block> cache=Nil;
  Bool shutdown = False;

  Unit run(){
    while(~ shutdown){
      await length(cache)<cacheSize;
```

```
      Bool next = await bs!hasNext();
      if (~next)
        shutdown=True;
      else      {
        Fut<Block> block = bs!nextBlock();
        Fut<Frame> frame = d!decode(block);
        Block store= await e!encode(frame);
        cache=appendright(cache,store);
      }
    }
  }

  Maybe<Block> nextBlock(){
    await ~ isEmpty(cache) || shutdown;
    Maybe<Block> retval=Nothing;
      case cache  {
        Cons(x,xs) => { cache=xs; retval= Just(x);}
        Nil => skip;
    }
    return retval;
  }

}

class RawDecoder implements Decoder{

  Frame decode(Fut<Block> f){
    Block block=f.get;
    return case block {
             Block(list) =>
               Frame(list);
           };
  }
}

class DiffEncoder implements Encoder{

  Block encode(Fut<Frame> f){
    Frame frame=f.get;
    List<Int> input=case frame { Frame(list) => list;};
    List<Int> result= Nil;
    Int prev=0;
    while (input != Nil){
      result=Cons(prev-head(input),result);
      input=tail(input);
    }
    return Block(reverse(result));
  }
}
```