Semi-Dynamic Session Types for ABS

TECHNISCHE UNIVERSITÄT DARMSTADT

Bachelor Thesis Proposal, Software Engineering Group
Anton Haubner

Contents

1 Motivation

Reasoning about distributed systems which employ asynchronous interactions between their components is usually difficult. For example, one might want to ensure, that said components adhere to a specific protocol, or that resources are properly allocated and released between uses.

When manually checking a model of a distributed system against a protocol specification becomes too labor-intensive and time-consuming, automatic verification can reduce the workload and help to avoid human errors.

Furthermore, if behavior can be dynamically added to the system during execution, for instance by an interface of the runtime, the set of properties which can be proven by static analysis of the system can become very limited.

Even in this case, some guarantees can still be enforced, if the schedulers of asynchronous processes are aware of a protocol and only schedule those tasks which do not violate it.

Thesis Topic

The Abstract Behavioral Specification language (ABS) is intended for modeling “distributed, object-oriented” software systems (Johnsen et al. [5]) and aims for “verifiable design” (Hähnle [4]) via a suite of external tools.

This thesis is based on (Kamburjan and Chen [6] and Kamburjan et al. [7]) and aims at developing a tool for automatically verifying whether objects within an ABS software model comply with a communication protocol, which is specified using Session Types.

Furthermore, it will extend all participating objects with a scheduler (Bjørk et al. [1]) which ensures, that only those processes are activated that the protocol permits to execute. This is necessary, since the ABS Model API (Schlatte et al. [8]) allows to asynchronously call methods during runtime, which will invalidate static verification. This idea has been explored in (Kamburjan et al. [7]).

Kamburjan and Chen [6] also introduce Async Dynamic Logic (ADL) such that ADL formulas can accompany protocol specifications and allow to express logical constraints on the program state before method invocations and after their termination. Although automatic verification of those constraints could be implemented using KeY-ABS (Din et al. [3]), it will likely exceed the time available for implementing the thesis. Instead, those constraints will optionally be implemented as runtime assertions, if time allows.

2 Working Packages

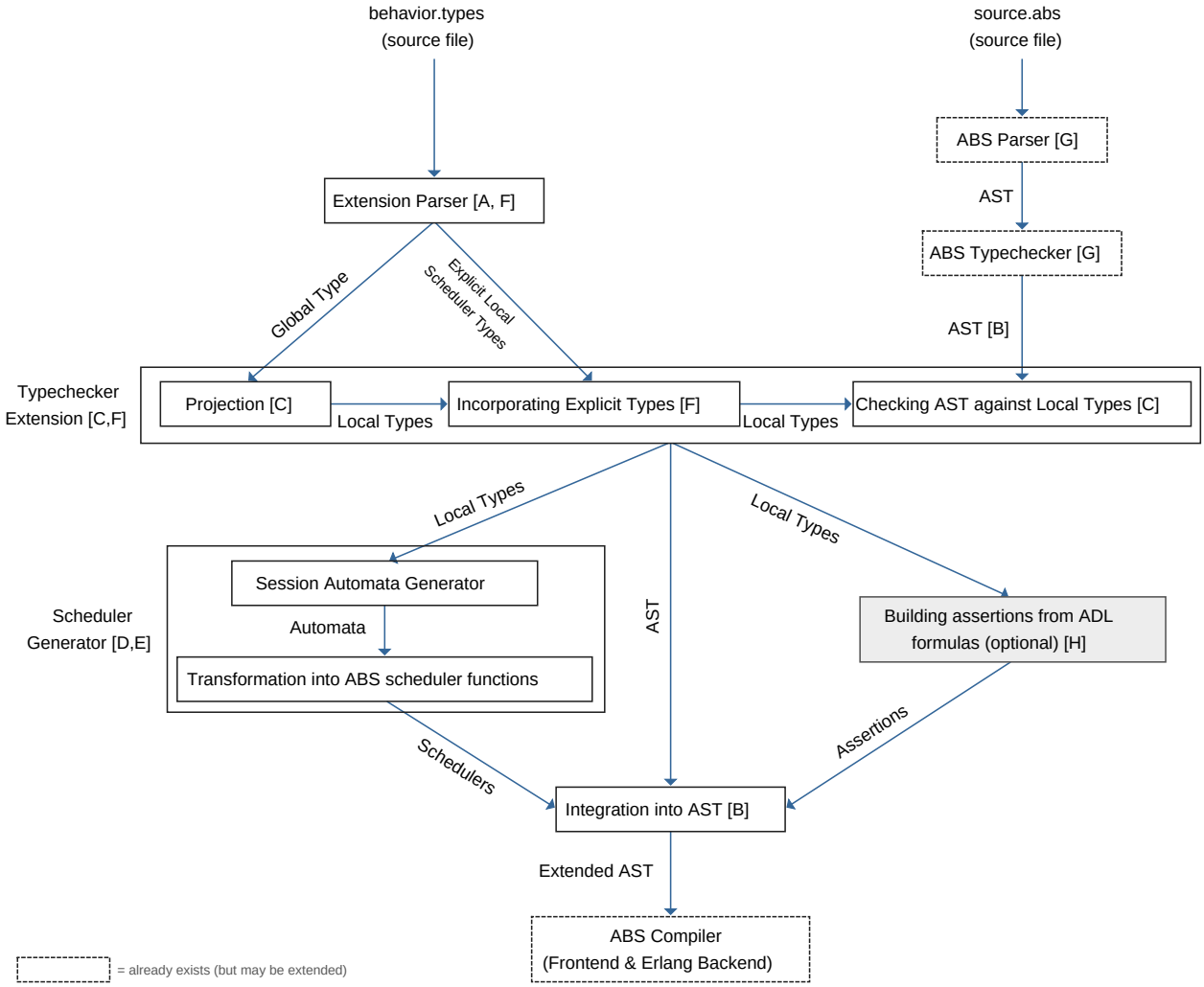


Figure 1: Expected internal structure of the verification tool. The approximate scope of the working packages has been annotated using the [·] notation.

All of the following working packages include a time investment proportional to the complexity of the package, which is reserved for conducting the necessary research using papers relevant to the topic. Also some time will be required for detailed planning of the implementation of the package and recording the results within the thesis document.

[A] Session Types syntax & parser The verification tool which is to be developed during the thesis will require an input source where developers can define their protocol specifications using a variant of Session Types.

Coupling between the ABS compiler and the verification tool shall be avoided as much as possible, such that no further complexity is introduced into the ABS compiler and the projects can be developed and maintained independently.

Therefore the verification tool shall parse its input from a file independent of the ABS source code of the software model.

This working package includes:

- deriving a practical syntax for defining Session Types from (Kamburjan and Chen [6]) and (Kamburjan et al. [7])
- developing a parser which provides the types as a data structure suitable for...

  - association with AST nodes of the ABS source file
  - processing by the type checker [C] (Projection, ...)
- this does not include syntax and parsing functionality for explicit local types, see working package [F].

[B] Passing AST from ABS parser to tool Most further working packages require the external types to be associated with the AST of the software model's ABS source code.

Code duplication should be avoided and therefore the AST generated by the ABS compiler shall be used, however as mentioned in [A], the ABS compiler shall be modified as little as possible.

Also some working packages ([E], [H]) entail modifying the AST. Therefore the potentially modified AST must be passed back to the ABS compiler such that it can be compiled into an executable form. The thesis only aims to create an AST compatible with the Erlang backend. This working package shall produce a tool with the following work flow:

1) parsing software model source code and applying Core ABS type checking to it using the ABS frontend.
2) invoking parser for external type specifications as specified in working package [A].
3) loading both data structures into the tool. After working packages [E] or [H] have been completed, this step will include enabling modifications to the AST using the external type definitions
4) passing (modified) AST back to ABS compiler to complete compilation

[C] Projection & type checking local action order For the most part, the external type definitions will specify a global protocol which needs to be *projected* on the individual units of the software model (objects, methods). This ultimately allows for type checking methods against the specification and generating schedulers. This working package includes:

- implementing projection
- implementing a type checker which statically ensures that the types are well-formed and the model is well-typed (Kamburjan and Chen [6])
  - this does not include checking for deadlocks or verifying ADL formulas embedded within the types (see [H])

[D] Extending ABS scheduler functionality The implementation of working package [E] requires, that schedulers are able to halt execution, if the execution of any available process would violate the specified protocol. They should continue execution, as soon as a viable process is available.

As of now, this is not supported by user defined schedulers in ABS, therefore it must be implemented as part of the thesis.

This working package includes:

- extending the ABS compiler backend to support scheduler functions which may not return a process to execute (support for `Maybe<...>` return type).

[E] Generating schedulers To ensure the protocol specified by the external type definitions is not violated at runtime due to calls via the Model API (Schlatte et al. [8]), scheduler functions (Bjørk et al. [1]) shall be generated from the type definitions for all objects participating in the protocol and integrated into the AST of the software model.

This working package includes:

- deriving Session Automata (Kamburjan et al. [7]) from the type definitions.
- transforming the Session Automata into pure scheduler functions (Bjørk et al. [1])
- validating, that the constructed scheduler functions indeed comply with the type specification
- integrating the schedulers into the AST provided by the ABS compiler

[F] Explicit local types As presented in (Kamburjan et al. [7]), types may explicitly be formulated for specific classes, not only as one global specification.

This can be used to specify valid behavior for an object across the whole protocol specification. For example, a file class shall only allow execution of its "read" method in between invocations of its "open" and "close" methods.

This working package includes:

- extending the syntax and parser of working package [A] with support for explicit local types

- extending the type checker of working package [C], such that it replaces projected local types with explicit local ones if explicit types have been defined for the class in question.

[G] Extending ABS Future wrapper Implementation of working package [E] requires, that futures from asynchronous method invocations may be saved without information on the wrapped return type. This is the case, since the generated schedulers will most likely need to store futures for multiple methods with different return types.

As of now, this is not supported by the `Fut<T>` type of ABS and therefore the compiler frontend and the (Erlang) backend have to be extended.

This working package includes:

- extending the ABS compiler frontend to support wrapping an unknown type "any" in Futures `Fut<T>`
- this includes modifying the parser and AST generation
- this does not include modifications to the (Erlang) backend compiler. The necessary adjustments to the backend compiler have already been implemented by the ABS developers.

[H] Optional: Generating assertions from ADL formulas As outlined in (Kamburjan and Chen [6]), the external type definitions may carry ADL formulas which describe constraints on the model's state during execution.

Those formulas shall be checked during execution of the model by extending the AST with assertion statements (Clarke et al. [2]) which implement the constraints described by the formula.

This working package includes:

- determining a transformation function from ADL formulas to ABS boolean expressions which may be placed in assertions.
- determining correct placement of the assertion statements within the software model's AST
- generating assertion statements and integrating them into the AST

3 Time Schedule

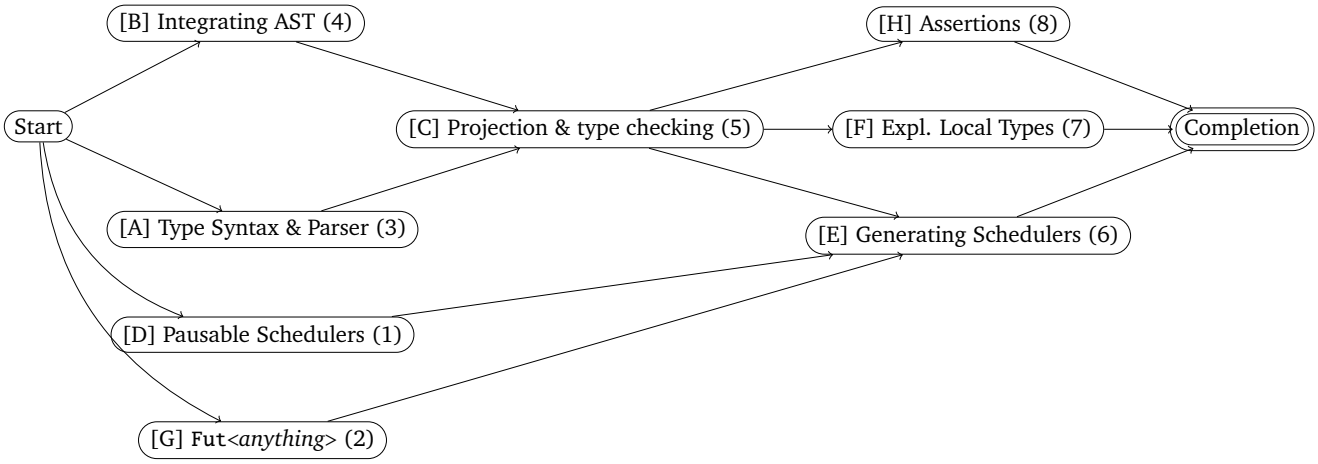
Available time

The following (theoretical) limits are to be kept in mind while scheduling the working packages:

A bachelor's thesis has to be written within 6 months. It should amount to about 12 credit points of work, which translate to $12\text{CP} \cdot 30\frac{\text{h}}{\text{CP}} = 360\text{h}$. Therefore, when assuming an even distribution of the work, this corresponds to about 14h per week.

Dependencies between Working Packages

The working packages outlined in 2 partially depend on each other. This has to be kept in mind while planning the order of implementation of the packages. The following diagram illustrates the dependencies between the working packages.



Based on this system of dependencies, the following implementation order has been determined.

(1) [D] Pausable Schedulers

(2) [G] `Fut<`*anything*`>`

(3) [A] Type Syntax & Parser

(4) [B] Integrating AST

(5) [C] Projection & type checking

(6) [E] Generating Schedulers

(7) [F] Expl. Local Types

(8) [H] Assertions

The following considerations have influenced the decided order:

- [D] and [G] have been picked as first items, such that necessary changes to Core ABS can be discussed with the ABS developers early on.

  This also allows to get familiar with the internals of the ABS compiler.

- Since [H] is optional, it has been placed at the end of the list.

Schedule

Available time: 26 weeks.

Working Package	Allotted Time	Start Date	End Date
Pausable Schedulers	2 weeks	08-05-19	21-05-19
`Fut<`*anything*`>`	1.5 weeks	22-05-19	01-06-19
Type Syntax & Parser	2 weeks	02-06-19	15-06-19
Integrating AST	1.5 weeks	16-06-19	26-06-19
Projection & type checking	5 weeks	27-06-19	31-07-19
Generating Schedulers (part 1)	2.5 weeks	01-08-19	18-08-19
Preparation for midway presentation	1 week	19-08-19	25-08-19
Generating Schedulers (part 2)	2.5 weeks	26-08-19	12-09-19
Expl. Local Types	3 weeks	13-09-19	03-10-19
Assertions	2.5 weeks	04-10-19	21-10-19
Final revisions, preparation of final presentation	2.5 weeks	22-10-19	08-11-19

Increased preparation effort for exams is to be expected around the time from the 8th of July to the 8th of August. This corresponds to about 8.5 weeks from the start of the thesis when assuming the 8th of May as starting point. Since around that time the working packages with the biggest time frames are scheduled, this should provide a slight safety margin to account for the heightened workload.

The implementation of the working package regarding generating schedulers [E] is split into two parts to accommodate for a week of preparation for the intermediate presentation which will probably take place around that time.

4 References

[1] Joakim Bjørk et al. “User-defined schedulers for real-time concurrent objects”. In: *Innovations in Systems and Software Engineering* 9.1 (2013), pp. 29–43.

[2] Dave Clarke et al. *Deliverable D1. 2 Full ABS Modeling Framework*. Tech. rep. HATS.

[3] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. “KeY-ABS: a deductive verification tool for the concurrent modelling language ABS”. In: *International Conference on Automated Deduction*. Springer. 2015, pp. 517–526.

[4] Reiner Hähnle. “The abstract behavioral specification language: a tutorial introduction”. In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2012, pp. 1–37.

[5] Einar Broch Johnsen et al. “ABS: A core language for abstract behavioral specification”. In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2010, pp. 142–164.

[6] Eduard Kamburjan and Tzu-Chun Chen. “Stateful behavioral types for ABS”. In: *arXiv preprint arXiv:1802.08492* (2018).

[7] Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. “Session-based compositional analysis for actor-based languages using futures”. In: *International Conference on Formal Engineering Methods*. Springer. 2016, pp. 296–312.

[8] Rudolf Schlatte et al. “Release the Beasts: When Formal Methods Meet Real World Data”. In: *It’s All About Coordination*. Springer, 2018, pp. 107–121.