Structuring the State and Behavior of ASMs: Introducing a Trait-Based Construct for Abstract State Machine Languages

Philipp Paulweber, Emmanuel Pescosta[⋆], and Uwe Zdun

University of Vienna
Faculty of Computer Science
Research Group Software Architecture
Währingerstraße 29, 1090 Vienna, Austria
{philipp.paulweber,uwe.zdun}@univie.ac.at

Abstract. Abstract State Machine (ASM) theory is a well-known state-based formal method to analyze, verify, and specify software and hardware systems. Nowadays, as in other state-based formal methods, the proposed specification languages for ASMs still lack easy-to-comprehend language constructs for type abstractions to describe reusable and maintainable specifications. Almost all built-in behaviors are implicitly defined inside a concrete ASM language implementation and thus, the behavior is hidden from the language user. In this paper, we present a new ASM syntax extension based on traits, which allows the specifier (language user) to define new type abstractions in the form of structure and behavior definitions to reuse, maintain, structure, and extend the functionality in ASM specifications. We describe the proposed language construct by defining its syntax and semantics. The decision to use a trait-based syntax extension over other object-oriented language constructs like interfaces or mixins was motivated and driven by the results of previously conducted empirical studies. Moreover, we outline details about the implementation of the trait-based syntax extension in our Corinthian Abstract State Machine (CASM) language implementation.

Keywords: Abstract State Machine, Trait, Structure, Modularization, CASM

1 Introduction

In 1993, Gurevich [1] introduced the Abstract State Machine (ASM) theory, which is a well-known state-based formal method consisting of transition rules and algebraic functions. It has been used extensively by scientists for a broad research field ranging from software and hardware to system engineering perspectives in order to specify, analyze, and verify systems in a formal way. ASMs are used to formally describe the evolution of function states in a step-by-step manner[1] and are used to describe sequential, parallel, concurrent, reflective, and even

[⋆] No affiliation. Member of CASM organization epescosta@casm-lang.org
[1] ASM theory was formerly called *Evolving Algebra*.

quantum algorithms. Based on the ASM theory by Gurevich [1], several theory improvements and ASM-based language implementations were developed, which were summarized by Börger and Stärk [2] and Börger and Raschke [3].

Prominent ASM languages and tools are AsmetaL [4], Corinthian Abstract State Machine (CASM) [5], and CoreASM [6]. Today, a common thread in the various ASM languages and tools, as well as in most other state-based formal methods, is that the proposed specification languages lack easy-to-comprehend abstractions to describe reusable and maintainable type specifications. While very few have embraced basic object-oriented abstractions such as classes and inheritance, more advanced type abstractions are usually missing. Therefore, in this paper we propose a new language construct for ASM specification languages to express type abstractions in the form of traits [7] to modularize specifications into structural state and behavioral parts.

2 Motivation

Modern object-oriented languages offer a variety of advanced type abstractions, and most offer either interfaces [8], mixins [9], or traits [7] in addition to classes and inheritance concepts. Interfaces establish a protocol and define method signatures to which a type has to conform [8]. They are often compared to a contract. Mixins define reusable behavior and structure that can be used to combine and form new types [9] [10]. Traits are similar to interfaces except that they can define stateless behavior which depends on the trait itself [11]. There is a heated debate in the object-oriented community[2], which of these abstractions is best suited to promote reusable and maintainable type specifications, and many implementations combine different language constructs to define type abstractions. A notable example would be the programming language Scala [12], which offers a trait syntax that is similar to the Java 8 [13] interface syntax and offers mixins type abstractions through the class-based implementation and extension syntax. Another example of mixed type abstraction concepts, namely interfaces and traits, can be found in the programming language Rust [14], where the language user has to express every interface definition through traits, and the types have to conform to specified traits and implement all required functionalities.

In the world of ASMs, only AsmL [15] has introduced an object model in the language through classes and interfaces to represent type abstractions, and to achieve structuring of the ASM specifications. Only the ASM implementation and language eXtensible ASM (XASM) by [16] has introduced a sub-ASM construct to achieve a component-based modularization approach. A more generic concept called *ambient* ASMs [3] introduces the possibility to achieve hierarchical state partitioning through nesting of context-sensitive (sub)program environments. Based on this state of the art, we started to investigate the introduction of a new type abstraction language construct in ASMs. But which language construct is suitable for ASMs to represent such type abstractions?

[2] See, e.g.: `https://stackoverflow.com/questions/925609`

Basically every language construct for forming type abstractions is suitable for ASMs, but it influences the understandability of the language considerably. For such an ASM extension, we consider the following properties important: (1) reuse and embed existing specifications; (2) describe built-in behavior of a language itself in the language; and (3) allow encapsulation of ASM states and corresponding behavior through modularization. Driven by the properties and questions raised, we conducted empirical studies to determine, which language construct – interfaces, mixins, or traits – is most understandable to ASM language users for expressing type abstractions [17]. The result of the experiments showed that the participants with strong object-oriented backgrounds (highly familiar with interfaces, not familiar with traits at all) had a similar to equal understanding of an interface and traits language construct in the experimental ASM syntax variants. Mixins, on the other hand, had a significantly lower understandability compared to traits and interfaces. Since the interface and traits type abstraction language constructs offer a similar to equal understandability, and novice language users seem to understand traits without even knowing the concept of traits, we investigated introducing traits into ASMs.

Moreover, the object-oriented communities often discuss traits more favorably than interfaces[3] and even point out that "Traits are Interfaces"[4] just with code-level reuse functionality. To gain a better understanding of how specifiers (language users) comprehend such trait-based specifications, we performed an eye-tracking experiment [17], where we observed the participants' gaze patterns. The results of this experiment showed that the participants could easily distinguish between behavioral and non-behavioral aspects of a given specification, when we applied our trait-based language construct to form state/behavior type abstractions.

3 A Trait-Based Construct for ASMs

This section proposes our trait-based language construct to extend the syntax of ASM specification languages. The syntax rules are defined and expressed in Backus Naur Form (BNF) (see Listing 1.1). The semantics of the proposed trait-based syntax extension is defined by lowering and transforming the new syntax elements to appropriate Turbo ASM [2] equivalent definitions (see example trait-based ASM Listing 1.2 and the transformed Turbo ASM Listing 1.3). The ASM specifications presented use the syntax of the CASM specification language[5]. The trait-based syntax extension is divided into three parts, namely *structural types*, *basic type behavior*, and *extended type behavior*.

In order to modularize the states (functions not classified as derived) in ASM, we introduce a *structural type* construct (see Listing 1.1, Line 2-4), which allows a language user to group one or multiple functions together (similar to members of an object-oriented class) to form a new *structure* type (see

[3] See, e.g.: `https://stackoverflow.com/questions/9205083`
[4] See, e.g.: `https://blog.rust-lang.org/2015/05/11/traits.html`
[5] For the CASM syntax description, see: `https://casm-lang.org/syntax`

StructureDefinition grammar rule). Each structure type defines a trait type through the defined state functions. The access to these functions is only allowed inside a proper basic behavior definition to clearly specify the access to an instantiated structure's state over dedicated behaviors (data encapsulation).

A *basic type behavior* (see Listing 1.1, Line 6-14) defines a set of rules and derived functions, which are associated with a certain domain type. We introduce a new ImplementDefinition to define a basic behavior consisting of one or more object-based derived function and/or rule definitions. The syntax for ObjectRuleDefinition and ObjectDerivedDefinition introduce a new keyword **this** as the first argument for all object-based rule and/or derived func-

```
 1 // Structural Types
 2 StructureDefinition ::= 'structure' Identifier '=' '{' ( FunctionDefinition )+ '}'.
 3 StructureLiteral    ::= [Type] '{' [Identifier ':' Term (',' Identifier ':' Term)*] '}'.
 4 Literal             ::= StructureLiteral | /* other literals */.
 5 // Basic Type Behavior
 6 ImplementDefinition     ::= 'implement' Identifier '=' '{'
 7                             ( ObjectRuleDefinition | ObjectDerivedDefinition )+ '}'.
 8 ObjectRuleDefinition    ::= 'rule' Identifier '(' 'this'
 9                             ( ',' Identifier ':' Type )* ')' [ '->' Type ] '=' Rule.
10 ObjectDerivedDefinition ::= 'derived' Identifier '(' 'this'
11                             ( ',' Identifier ':' Type )* ')' '->' Type '=' Term.
12 MethodCall              ::= Term '.' Identifier ['(' Term (',' Term)* ')'].
13 CallRule                ::= MethodCall | ( Identifier [ '(' Term (',' Term)* ')' ] ).
14 Term                    ::= MethodCall | 'this'
15 // Extended Type Behavior
16 BehaviorDefinition       ::= 'behavior' Identifier '=' '{'
17                              ( ObjectRuleDeclaration | ObjectDerivedDeclaration
18                              | ObjectRuleDefinition  | ObjectDerivedDefinition )+ '}'.
19 ImplementForDefinition   ::= 'implement' Identifier 'for' Identifier '=' '{'
20                              ( ObjectRuleDefinition  | ObjectDerivedDefinition )+ '}'.
21 ObjectRuleDeclaration    ::= 'rule'    Identifier ':' 'Object' ('*' Type)* '->' Type.
22 ObjectDerivedDeclaration ::= 'derived' Identifier ':' 'Object' ('*' Type)* '->' Type.
```

Listing 1.1: Trait-Based ASM Syntax Extension

```
structure X = {
  function f1 : -> Integer
  function f2 : Integer -> Boolean
}






rule R1 =
  let v1 = X{ f1: 1,
    f2: ( 2 ) -> false } in skip
implement X = {
  derived d1( this ) -> Integer =
    this.f1

  rule R2( this, a1 : Integer ) =
    if a1 > -5 and this.d1 < 5 then
      this.f2( a1 ) := true
}
behavior Y = {
  derived d2 : Object -> Integer

  derived d3( this ) -> Boolean
    = this.d2 * this.d2 > 100
}
implement Y for X = {
  derived d2(this) -> Integer = this.f1
}
// ...
```

Listing 1.2: Trait-Based ASM

```
domain X
function X_f1 : X -> Integer
function X_f2 : X * Integer -> Boolean
rule X_instantiate( a1 : Integer
, a2 : Integer -> Boolean ) -> X =
  let object = new X in {
    X_f1( object ) := a1
    X_f2( object ) := a2
    result := object
  }
rule R1 =
  let v1 = X_instantiate( 1,
    { ( 2 ) -> false } ) in skip

derived X_d1( this : X ) -> Integer =
  X_f1( this )

rule X_R2( this : X, a1 : Integer ) =
    if a1 > -5 and X_d1(this) < 5 then
      X_f2( this, a1 ) := true




derived X_d3( this : X ) -> Boolean
    = X_d2( this ) * X_d2( this ) > 100
}

derived X_d2(this:X) -> Integer = X_f1(this)

// ...
```

Listing 1.3: Turbo ASM Equivalent

tion definitions. The type of the argument variable `this` equals the type of the `ImplementDefinition` and it enables the access to the domain's or structure's behavior. The access happens through a `MethodCall` syntax, which uses a dot operator between a term, a target name, and a non-negative arity of arguments. The target name can be a function name or a rule name.

An *extended type behavior* (see Listing 1.1, Line 16-22) defines a set of rules and derived functions, and forms a new type in the type system. If a domain and/or structural type wants to use the functionality, it has to implement the extended behavior. The `BehaviorDefinition` defines an explicit trait with type name consisting of zero or more `ObjectRuleDeclaration` rule names and/or `ObjectDerivedDeclaration` derived function names. Please note that for all object-based declarations we introduced a generic `Object` argument type at the first position. The `Object` type gets checked against the domain or structural type which is implementing this declared behavior. A specifier can use the `Object` type for any other argument or target type in a declaration. Additionally, a trait can define a default behavior through zero or more `ObjectRuleDefinition` rule names and/or `ObjectDerivedDefinition` derived function names, which depends only on the functionality of the trait itself. Each domain and/or structural type that wants to support a certain behavior has to specify an `ImplementForDefinition` and provide the missing definitions of the trait declarations. If the trait defines a default behavior, the domain and/or structural type inherits this definition. This enables code reuse capabilities.

Listing 1.2 depicts an example trait-based ASM specification using all new syntax grammar rules and Listing 1.3 depicts the equivalent semantics-preserving Turbo ASM specification. The proposed trait-based syntax extension is realized in our CASM language implementation[6]. In order to provide a clean solution, we updated our CASM language front-end implementation and introduced two new internal Abstract Syntax Tree (AST) representations before the specification gets transformed to the CASM Intermediate Representation (CASM-IR) [5].

By introducing the proposed trait-based construct, we were able to explicitly specify the behavior of the CASM language itself in CASM in the form of a *prelude*[6] specification, which gets automatically loaded (imported) for every parsed CASM specification. Each functionality of the CASM language (e.g. operators) is mapped to a behavior (trait) in the prelude specification. The language user can explore and extend the behaviors of CASM in CASM. Moreover, the prelude specification reduced the complexity of the CASM implementation.

4 Conclusion

In this paper, we present a trait-based construct for ASM languages. It allows to specify composable models through the usage of domain and structural type objects, where the behavior can be defined and implemented in a reusable manner. The modularization and composing of object-oriented models is achieved by

[6] For sources, see: `https://github.com/casm-lang/libcasm-fe/pull/205`

specifying structural states along with their behaviors clearly separated through traits. Novel about this contribution is that ASM language users can directly define the semantics of operations over domain (structure) types through this trait-based construct in the ASM language itself. To clearly separate structure and behavior, we only allow the definition of modifications to structural objects through a proper behavior definition. Based on previously conducted empirical studies, the current state of the art, and our current proposed trait-based construct, we believe that this is the first step towards clearer and more understandable ASM specifications by separating the structural (state) and behavioral elements through dedicated definitions.

References

[1] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide - Specification and Validation Methods," pp. 9–36, New York, NY, USA: Oxford University Press, Inc., 1995.

[2] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer Science & Business Media, 2003.

[3] E. Börger and A. Raschke, *Modeling Companion for Software Practitioners.* Springer, 2018.

[4] A. Gargantini, E. Riccobene, and P. Scandurra, "A metamodel-based language and a simulation engine for abstract state machines," *Journal of Universal Computer Science*, vol. 14, no. 12, pp. 1949–1983, 2008.

[5] P. Paulweber, E. Pescosta, and U. Zdun, "CASM-IR: Uniform ASM-Based Intermediate Representation for Model Specification, Execution, and Transformation," in *ABZ 2018*, LNCS 10817, pp. 39–54, Springer, 2018.

[6] R. Farahbod, V. Gervasi, and U. Glässer, "CoreASM: An Extensible ASM Execution Engine," *Fundamenta Informaticae*, vol. 77, no. 1-2, pp. 71–104, 2007.

[7] G. Curry, L. Baer, D. Lipkie, and B. Lee, "Traits: An approach to multiple-inheritance subclassing," in *Proceedings of the SIGOA Conference on Office Information Systems*, (New York, NY, USA), pp. 1–9, ACM, 1982.

[8] P. S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff, "Interfaces for strongly-typed object-oriented programming," in *OOPSLA*, pp. 457–467, ACM, 1989.

[9] M. Flatt, S. Krishnamurthi, and M. Felleisen, "Classes and mixins," in *ACM SIGPLAN-SIGACT POPL*, (New York, NY, USA), pp. 171–183, ACM, 1998.

[10] G. Bracha and W. Cook, "Mixin-based inheritance," *ACM Sigplan Notices*, vol. 25, no. 10, pp. 303–311, 1990.

[11] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable Units of Behaviour," in *ECOOP*, pp. 248–274, Springer, 2003.

[12] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala.* Artima Inc, 2008.

[13] A. Potts and D. H. Friedel, *Java programming language handbook.* Coriolis Group Books, 2018.

[14] N. D. Matsakis and F. S. Klock II, "The Rust Language," in *ACM SIGAda Ada Letters*, vol. 34, pp. 103–104, ACM, 2014.

[15] Y. Gurevich, B. Rossman, and W. Schulte, "Semantic Essence of AsmL," in *Formal Methods for Components and Objects*, pp. 240–259, Springer, 2004.

[16] M. Anlauff, "XASM – An Extensible, Component-based Abstract State Machines Language," in *ASM - Theory and Applications*, pp. 69–90, Springer, 2000.

[17] G. Simhandl, P. Paulweber, and U. Zdun, "Design of an Executable Specification Language Using Eye Tracking," in *EMIP'19 (at ICSE'19)*, May 2019.