Rules for Verification of Interactive Systems

Marcelo de Almeida Maia
marcmaia@dcc.ufmg.br
Universidade Federal de Ouro Preto

Roberto da Silva Bigonha
bigonha@dcc.ufmg.br
Universidade Federal de Minas Gerais

Abstract

Recently, we have proposed an interaction based extension[4] to the formal method ASM[2]. The extension provides mechanisms to specify directly how pieces of specification can interact with each other. The goals of the extension are manifold, ranging, for example, from the modularization of a specification to the specification of mobile objects. In this work, we propose a type discipline for the Interactive ASMs focused on the interactive portion of the specification.

1 Introduction

We have proposed in [4] a method which provides a suitable way to specify dynamic interconnection of specification units and the rules that guide their interaction. This approach, the Interactive Abstract State Machines (IASMs), delivers enough flexibility to address problems such as specifying concurrent, mobile and large software systems. In order to provide mechanisms that enable a productive specification writing within the context of an automated tool and also help the partial checking of the specifications, we introduce in this work a type discipline to the Interactive Abstract State Machines. Instead of providing a whole type system for IASMs, this work will focus only on the interactive portion of the specification.

An influential work in the semantics of interactive systems is π-calculus[5], where the channels of communication can be transmitted over other channels, allowing a process to acquire dynamically new channels, and thus expressing, for example, mobility. Some type systems have already been proposed for the π-calculus, starting with the Milner's sort discipline based on the polyadic π-calculus[5], a subtyping extension[6], an extension with parametric polymorphism[7]. Recently, Cardelli and Gordon introduced the mobile ambients, as a new approach to mobility, and they have already proposed a type system[1] with aims similar as ours.

Our intention is to bring to the IASM context the same range of benefits delivered by typed calculi, even though our context is not based on any calculus. In the following sections we review the IASM method and present its type system.

S	::=	main $U_1 \ldots U_m$ $Inst_1 \cdots Inst_n$ $C_1 \cdots C_p$	$(m,n,p \in \aleph)$
$Inst$	::=	$u_i : U_j$	$(i \in \{1..n\}, j \in \{1..m\})$
C	::=	u_r <-> u_s	$(r,s \in \{1..n\}$ and $r \neq s)$

Figure 1: System Specification

U	::=	unit *unit_name* function names *f_n* interaction *i* rules *rules*

Figure 2: Unit Definition

2 The Interactive Abstract State Machine Language

An IASM specification is strongly based on distributed ASM. For the sake of better understanding, we suggest to the reader the work in [3], where we briefly present ASM and propose an interesting case study on mobile objects. An IASM specification is defined as a set of unit definitions, static unit instances, and static unit connections. A unit definition contains an internal state, an interaction rule which describes how that unit interacts with the others, and an internal rule which describes the local computation of the unit.

The abstract syntax of a system specification S is defined as in Figure 1, where U_i, $i \in 1..m$, is a unit definition, $Inst_i$, $i \in 1..n$, is a static unit instance declaration, and C_i, $i \in 1..p$, is a static unit connection defined with the operator <->, which updates, inside each corresponding unit, the locations referring to the other unit.

A unit definition U is defined as in Figure 2. The function names *f_n* are in a subset of the vocabulary, defining locations representing a local state alterable only by the local unit rules and interaction.

The abstract syntax for an interaction i is defined in Figure 3. The basic operators for interaction are those that provide input and output within a unit, respectively, <- and ->, which are used to receive a value from a unit into a variable and to send a value to a unit. The operator <-- denotes a buffered input that avoids an inconsistent update if two or more different inputs to the same variable occur in the same step. Since we expect to define dynamically the communication topology, we provide the connect operator which binds a unit name to some unit instance. The operators *new* and *destroy* are used to create and destroy unit instances. In order

i	::=	*internal_pub_name* -> *u_name*
	|	*buffered_var* <-- *u_name.pub_name*
	|	*var* <- *u_name.pub_name*
	|	connect *unit* : *U.s* | connect *unit* : *U* | connect *u*
	|	new *unit* : *U*
	|	destroy *unit* : *U*
	|	$i + i$ | $i \mid i$ | $i ; i$ | $i : l$
	|	waiting(*name*)
	|	if *guard* then i

Figure 3: Interaction Rule

to address complex interaction patterns that may exist between units we provide the well-known composition operators "+" (non-deterministic choice), "|" (parallel composition), and ";" (sequential composition). In order to synchronize the interaction part with the computation part of a unit we introduce labeled interactions and the barrier waiting*(name)*. The label l uniquely identifies an interaction, and denotes how many times the interaction labeled with l has completely occurred. Its initial state is *zero*.

Finally, the internal rules *rules* are defined just as traditional ASM rules. These rules work by changing the internal state represented by *function_names*.

3 A Type System for Interactions

In this section, we will define a type system focusing on the interaction between the units. We will assume that the elements of the super-universe are organized within ground types K (possibly the type of a defined unit). The possibility of more complex types should be addressed elsewhere.

In usual type systems specifications, the environment contains the types of free variables during the processing of programs fragments. Since we are interested in checking the interaction between units, we will enrich the contextual information with the occurrence of fragments of interaction rules.

Thus, the environment Γ will be a list of contextual information items: $\emptyset, \zeta_1, \ldots, \zeta_{|\Gamma|}$, where each ζ_i can be: 1) a unit A that has been defined, 2) a type information $A.x : K$ (x inside unit A has type K), 3) a static connection $A \rightleftharpoons B$ (A statically connects B, and vice versa), 4) an input fragment $B.(y<\text{-}A.x)$ (unit B has a rule that receives a value from the location x inside unit A into the location y), 5) an output fragment $B.(y\text{->}A)$ (unit B has a rule that sends the contents of the location y to unit A), 6) a dynamic connection fragment $B.(connect\ a : A)$ (unit B has a rule that connects to unit A).

Now, we present the type system rules. We start with the set of kind of judgments used in the rules. The judgments are described in Figure 4. We classify the type system rules based on the possible kinds of judgments.

So, the first class of type rules, shown in Figure 5, checks if an environment is well-formed. Each environment rule specifies what kind of item can be present in the environment.

Another class of rules is the one which checks if a type is well-formed. Since we will have only the ground types Bool and Int, and the types of the defined units, there are just three rules for types, and they are shown in Figure 6.

The third class of rules, shown in Figure 7, is the one which checks if a declaration is well-formed, where the first three rules are used to start the checking process of a specification. Below, we define the elements used in those rules:

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash D$	D is a well-formed declaration in Γ
$\Gamma \vdash I$	I is a well-formed interaction in Γ

Figure 4: Judgments

$$\frac{}{\emptyset \vdash \diamond}\ (\text{Env } \emptyset)\ (1.1) \qquad \frac{\Gamma \vdash K \quad x \notin dom(\Gamma)}{\Gamma, x : K \vdash \diamond}\ (\text{Env } x)\ (1.2) \qquad \frac{U \notin dom(\Gamma)}{\Gamma, U \vdash \diamond}\ (\text{Env } U)\ (1.3)$$

$$\frac{\Gamma \vdash K \quad A.x \notin dom(\Gamma)}{\Gamma, A.x : K \vdash \diamond}\ (\text{Env } A.x)\ (1.4) \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma, A \rightleftharpoons B \vdash \diamond}\ (\text{Env } A \rightleftharpoons B)\ (1.5)$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma, A.(x\text{->}B) \vdash \diamond}\ (\text{Env } A.(x\text{->}B))\ (1.6) \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma, A.(x\text{<-}B) \vdash \diamond}\ (\text{Env } A.(x\text{<-}B.y))\ (1.7)$$

$$\frac{\Gamma \vdash A}{\Gamma, A.(\textit{connect } \square) \vdash \diamond}\ (\text{Env } A.(\textit{connect } \square))\ (1.8) \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma, A.(\textit{connect } B) \vdash \diamond}\ (\text{Env } A.(\textit{connect } B))\ (1.9)$$

Figure 5: Environment

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Bool}}\ (\text{Type } \mathit{Bool})\ (2.1) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Int}}\ (\text{Type } \mathit{Int})\ (2.2) \qquad \frac{\Gamma, U, \Gamma' \vdash \diamond}{\Gamma, U, \Gamma' \vdash U}\ (\text{Type } \mathit{Unit}\ U)\ (2.3)$$

Figure 6: Types

- $U_i\ i \in 1..m$ is a unit definition represented below as *unit* $U \equiv D\ I$, where D is a list $x_i : K_l\ ^{i=1..l}$, where l is the number of internal declarations and I is the declaration of the interaction pattern.
- $I_i\ i \in 1..n$ is a static instance declaration represented as $u : U$, where U is an already defined unit.
- $C_i\ i \in 1..r$ is a static connection declaration represented as u_i <-> u_j, where u_i and u_j are already declared static units and $i \neq j$.
- *Env* is a function that returns a list with the static environment generated by its corresponding parameters, which are unit definitions, and/or static declarations of instances or connections. We also assume, that if we seek for an information of kind $A \leftrightharpoons B$, then if we find $B \leftrightharpoons A$, a successful match occurs. Besides this we introduce a matching operator for connection items $A \leftrightarrows B$, which seeks for *A.connect* $\square$, or *A.connect* B or $A \leftrightharpoons B$. *Env* is defined in Figure 8

Finally, in Figure 9 we present the rules used when checking the interaction between units. We use the abbreviation *conn* for *connect*.

In Proposition 1 appears some characteristics of the above rules. We will show its proof elsewhere.

Proposition 1 *If a system specification is well-formed then: 1) for every input interaction there exists the corresponding output and vice versa; 2) for every input or output interaction there exists a connection interaction fragment, such that, if performed, it enables the interaction; 3) the function names that hold an exchanged value have an equivalent type in the corresponding pair input/output interaction; 4) for every static connection there exists one, and only one, corresponding function name to be updated with the corresponding unit instance name; 5) for every dynamic connection there exists a counterpart fragment which may commit the connection.*

$$\frac{\Gamma, Env(U_2, ..., U_m, C_1, ..., C_r) \vdash U_1 \quad ... \quad \Gamma, Env(U_1, ..., U_{m-1}, C_1, ..., C_r) \vdash U_m}{\Gamma \vdash \text{main } U_1 \ ... \ U_m \ I_1 \ ... \ I_n \ C_1 \ ... \ C_r}\ (\text{Decl } \textit{Specification})\ (\textit{Unit definition checking})\ (3.1)$$

$$\frac{\Gamma, U_1, ..., U_m \vdash I_1 \quad ... \quad \Gamma, U_1, ..., U_m \vdash I_n}{\Gamma \vdash \text{main } U_1 \ ... \ U_m \ I_1 \ ... \ I_n \ C_1 \ ... \ C_r}\ (\text{Decl } \textit{Specification})\ (\textit{Static instance declaration checking})\ (3.2)$$

$$\frac{\Gamma, Env(U_1, ..., U_m, I_1, ..., I_n) \vdash C_1 \quad ... \quad \Gamma, Env(U_1, ..., U_m, I_1, ..., I_n) \vdash C_r}{\Gamma \vdash \text{main } U_1 \ ... \ U_m \ I_1 \ ... \ I_n \ C_1 \ ... \ C_r}\ (\text{Decl } \textit{Specification})\ (\textit{Static connection declaration checking})\ (3.3)$$

$$\frac{\Gamma, Env(D) \vdash U.I \quad \Gamma \vdash D}{\Gamma \vdash \text{unit } U \equiv D\ I}\ (\text{Decl } \textit{Unit})\ (3.4) \qquad \frac{\Gamma \vdash E : A \quad \Gamma \vdash A}{\Gamma \vdash I : A \text{ initially } E}\ (\text{Decl } \textit{Internal State})\ (3.5)$$

$$\frac{\Gamma \vdash K_1 \quad \Gamma \vdash K_{|D|}}{\Gamma \vdash I_1 : K_1, ..., I_{|D|} : K_{|D|}}\ (\text{Decl } \textit{Static or Internal State})\ (3.6)$$

$$\frac{\Gamma', a : A, b : B, A.(b : B), B.(a : A), \Gamma'' \vdash \diamond}{\Gamma', a : A, b : B, A.(b : B), B.(a : A), \Gamma'' \vdash a\text{<->}b}\ (\text{Decl } \textit{Static Connection})\ (3.7)$$

$$\nexists B.(a' : A), A.(b' : B), A.(\textit{connect } b : B), B.(\textit{connect } a : A) \mid B.a' \neq B.a \wedge A.b' \neq A.b$$

Figure 7: Rules for Declarations

$Env(\delta_1, ..., \delta_x) = Env(\delta_1), ..., Env(\delta_x)$
$Env(\textit{unit } U \equiv D\ I) = U, Env_L(U, D), Env_L(U, I)$
$Env(u : U) = u : U$
$Env(u_i$ <-> $u_j) = U_i \rightleftharpoons U_j$, where $u_i : U_i$ and $u_j : U_j$.
$Env_L(U, x_1 : K_1, ..., x_l : K_l) = U.x_1 : K_1, ..., U.x_l : K_l$
$Env_L(U, x\text{->}A) = U.(x\text{->}A)$
$Env_L(U, x\text{<-}A.y) = U.(x\text{<-}A.y)$
$Env_L(U, \textit{connect } u) = U.\textit{connect } \square$
$Env_L(U_1, \textit{connect } u : U_2) = U_1.\textit{connect } U_2$
$Env_L(U, \bullet I) = Env_L(U, I)$, where $\bullet$ is a unary interaction rule constructor (eg. if-then).
$Env_L(U, I_1 \circ I_2) = Env_L(U, I_1), Env_L(U, I_2)$, where $\circ$ is a binary interaction rule constructor (eg. I ; I).

Figure 8: Function *Env*: computes the environment

$$\frac{\Gamma', x : \sigma, B.y : \sigma, B.(y\text{->}A), A \leftrightarrows B, \Gamma'' \vdash \diamond}{\Gamma', x : \sigma, B.y : \sigma, B.(y\text{->}A), A \leftrightarrows B, \Gamma'' \vdash A.(x \text{ <- } B.y)}\ (\text{Interaction } \textit{Input})\ (4.1) \qquad \frac{\Gamma', x : \sigma, B.y : \sigma, B.(y\text{<-}A.x), A \leftrightarrows B, \Gamma'' \vdash \diamond}{\Gamma', x : \sigma, B.y : \sigma, B.(y\text{<-}A.x), A \leftrightarrows B, \Gamma'' \vdash A.(x \text{ -> } B)}\ (\text{Interaction } \textit{Output})\ (4.2)$$

$$\frac{\Gamma', B.(\textit{conn } u), \Gamma'' \vdash \diamond}{\Gamma', B.(\textit{conn } u), \Gamma'' \vdash A.(\textit{conn } \square)}\ (\text{Interaction } \textit{Conn})\ (1)\ (4.3) \qquad \frac{\Gamma', B.(\textit{conn } A), \Gamma'' \vdash \diamond}{\Gamma', B.(\textit{conn } A), \Gamma'' \vdash A.(\textit{conn } \square)}\ (\text{Interaction } \textit{Conn})\ (2)\ (4.4) \qquad \frac{\Gamma', B.(\textit{conn } u), \Gamma'' \vdash \diamond}{\Gamma', B.(\textit{conn } \square), \Gamma'' \vdash A.(\textit{conn } B)}\ (\text{Interaction } \textit{Conn})\ (3)\ (4.5)$$

$$\frac{\Gamma', B.(\textit{conn } u : A), \Gamma'' \vdash \diamond}{\Gamma', B.(\textit{conn } A), \Gamma'' \vdash A.(\textit{conn } B)}\ (\text{Interaction } \textit{Conn})\ (4)\ (4.6) \qquad \frac{\Gamma \vdash A.(u : U)}{\Gamma \vdash A.(\textit{new } u : U)}\ (\text{Interaction } \textit{new})\ (4.7) \qquad \frac{\Gamma \vdash A.(u : U)}{\Gamma \vdash A.(\textit{destroy } u : U)}\ (\text{Interaction } \textit{destroy})\ (4.8)$$

$$\frac{\Gamma \vdash g : bool \quad \Gamma \vdash A.i}{\Gamma \vdash A.(\textit{if } g \textit{ then } i)}\ (\text{Interaction } \textit{if})\ (4.9) \qquad \frac{\Gamma \vdash A.(i_1) \quad \Gamma \vdash A.(i_2)}{\Gamma \vdash A.(i_1 ; i_2)}\ (\text{Interaction } ;)\ (4.10) \qquad \frac{\Gamma \vdash A.(i_1) \quad \Gamma \vdash A.(i_2)}{\Gamma \vdash A.(i_1 \mid i_2)}\ (\text{Interaction } \mid)\ (4.11)$$

$$\frac{\Gamma \vdash A.(i_1) \quad \Gamma \vdash A.(i_2)}{\Gamma \vdash A.(i_1 + i_2)}\ (\text{Interaction } +)\ (4.12) \qquad \frac{\Gamma \vdash A.(i : l)}{\Gamma \vdash A.(i : l)}\ (\text{Interaction } :)\ (4.13) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash A.(\textit{waiting}(var))}\ (\text{Interaction } \textit{waiting(var)})\ (4.14)$$

Figure 9: Rules for Interaction Checking

4 Conclusions

The type system presented in this work performs a more general checking since it checks not only the types of the information exchanging rules, but also the consistency between the interaction rules of the unit definitions. This checking can be very useful when debugging large specifications. It seems to us that it would be unreasonable trying to specify a very large system without any kind of automatic checking. It is easy to see that all the type checking presented in this work can be made statically.

Additionally, this type system also provide useful insights to better understand the IASM semantics. And since we have used a standard technique to describe the checking system, we hope that any comparison with other approaches can be made easily.

There are still some worthwhile issues for further studies, such as the connection of the presented type rules with a type system for the whole specification, which would include, for example, subtyping. We should also reason about the type soundness within the system. We should still separate the notions of interchecking and intrachecking. This separation is essential for separate checking of specifications, which is specially important because ASM and IASM specification are suitable for execution, and we want to support some specification libraries.

References

[1] L. Cardelli and A. Gordon. Types for mobile ambients. In *26th Annual ACM Symposium on Principles of Programming Languages*, January 1999.

[2] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[3] M. Maia and R. Bigonha. Interaction based semantics for mobile objects. This volume, 1999.

[4] M. Maia, V. Iorio, and R. Bigonha. Interacting Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*, 1998.

[5] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, Oct. 1991.

[6] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 1996.

[7] B. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *Principles of Programming Languages (POPL)*, 1997.