A New Coordination Language MediatE

Yi Li and Meng Sun

LMAM and Department of Informatics, School of Mathematical Sciences, Peking University, Beijing, China
`liyi_math@pku.edu.cn, sunmeng@math.pku.edu.cn`

Abstract. tbd

1 Introduction

2 Overview

3 The Grammar

In this section, we mainly focus on the syntax of our language, it is divided into different parts and basically described in *Extended Backus-Naur form (EBNF)*.

Let's introduce the overview of this language first.

$$
\begin{aligned}
\langle \textit{program} \rangle &::= [\ \langle \textit{statement} \rangle\ ]^* \\
\langle \textit{statement} \rangle &::= \langle \textit{importStmt} \rangle \mid \langle \textit{typedefStmt} \rangle \mid \langle \textit{functionStmt} \rangle \\
&\quad \mid \ \langle \textit{channelStmt} \rangle \mid \langle \textit{componentStmt} \rangle \\
&\quad \mid \ \langle \textit{connectorStmt} \rangle
\end{aligned}
$$

3.1 Type System

The basic idea behind this type system is that we try to satisfy both researchers and programmers.

$$
\begin{aligned}
\langle \textit{primitiveType} \rangle &::= \text{`}\textbf{int}\text{'}\ [\ \langle \textit{term} \rangle\ \text{`}\textbf{..}\text{'}\ \langle \textit{term} \rangle\ ] \\
&\quad \mid \ \text{`}\textbf{double}\text{'} \mid \text{`}\textbf{char}\text{'} \mid \text{`}\textbf{bool}\text{'} \mid \text{`}\textbf{NULL}\text{'} \\
&\quad \mid \ \text{`}\textbf{enum}\text{'}\ \{\ \langle \textit{identifier} \rangle\ ^+\ \text{`}\mathbf{\}}\text{'} \\
&\quad \mid \ \langle \textit{identifier} \rangle \\
\langle \textit{extendedType} \rangle &::= \langle \textit{primitiveType} \rangle \\
&\quad \mid \ \langle \textit{type} \rangle\ \text{`}|\text{'}\ \langle \textit{type} \rangle \\
&\quad \mid \ \text{`}\textbf{array}\text{'}\ \langle \textit{extendedType} \rangle\ \text{`['}\ \langle \textit{term} \rangle\ \text{`]'} \\
&\quad \mid \ \text{`}\textbf{map}\text{'}\ \text{`['}\ \langle \textit{extendedType} \rangle\ \text{`]'}\ \langle \textit{type} \rangle \\
&\quad \mid \ \text{`}\textbf{struct}\text{'}\ \text{`}\mathbf{\{}\text{'}\ (\ \langle \textit{identifier} \rangle\ \text{`}\mathbf{:}\text{'}\ \langle \textit{type} \rangle\ )^+\ \text{`}\mathbf{\}}\text{'}
\end{aligned}
$$

Definition 1 (Finite Types). *A type in MediatE is* finite *iff. it has a finite value domain.*

Primitive Type. MediatE provides built-in support for common primitive types, they are,

- *Int.* Similar to
- *Double.*
- *Bool.* Boolean variables.
- *Enum.* An *enumeration* is a finite collection of unique identifiers.
- *NULL.* For simplicity, in MediatE we don't have pointers references, however, sometimes a non-sense value is strongly required to denote uninitialized value, illegal operation (e.g. divided by zero) and exceptions. Consequently, we define a single-value type NULL, of which the only possible value is *null*.

Extended Type. Extended type offers an approach to contruct complex data types with simpler ones. Four extending patterns are introduced as follows,

- *Union.* The *union* operator '|' is designed to combine two *disjoint* types as a more complicated one. This is similar to the union type in C language but much easier to use.
- *Array* and *Slice.* An *array* T[n] is a finite ordered collection containing exactly n elements of type T. Moreover, a *slice* is an array of which the capacity is not specified.
- *Map.* A *map* $[T_{key}]\ T_{val}$ is a dictionary that maps a key of type T_{key} to a value of type T_{val}.
- *Struct.* A *struct* $\{field_1 : T_1, \cdots, field_n : T_n\}$ contain n fields, each has a particular type T_i and a unique identifier id_i.

Type systems often differ greatly between formal models and real-world programming languages. For example, PRISM[2], ..

In MediatE's type system, most of basic types are finite except for the unlimited integers and doubles. Besides, extended types are also finite if they are based on finite ones.

Definition 2 (Subtyping). *Type T_1 is a subtype of T_2, denoted by $T_1 \preccurlyeq T_2$, iff. a value of type T_1 can be converted to a value of type T_2 losslessly.*

$$\frac{l_1, u_1, l_2, u_2 \in \mathbb{Z}, l_1 \leq u_1 \wedge l_2 \geq u_2}{\text{int } l_1 \text{ .. } u_1 \preccurlyeq \text{int } l_2 \text{ .. } u_2} \text{ S-FiniteInt}$$

$$\frac{n \in \mathbb{Z}}{\text{enun } \{ \ item_0, \cdots, item_{n-1}\} \preccurlyeq \text{int } 0 \text{ .. (n - 1)}} \text{ S-Enum}$$

$$\frac{\cdot}{\text{bool} \preccurlyeq \text{int } 0 \text{ .. } 1} \text{ S-Bool}$$

$$\frac{l, u \in \mathbb{Z}}{\text{int } l \text{ .. } u \preccurlyeq \text{int}} \text{ S-Int}$$

$$\frac{T_1 \preccurlyeq T_3, T_2 \preccurlyeq T_4}{T_1|T_2 \preccurlyeq T_3|T_4} \text{ S-Union}$$

3.2 Type Alias and Concrete Types

⟨*typedefStmt*⟩ ::= ‘**typedef**’ ⟨*type*⟩ ‘**as**’ ⟨*identifier*⟩ [‘**init**’ ⟨*term*⟩]

In most programming languages, a non-reference type has a built-in default value. However, such values are not specified by programmers, and sometimes may lead to unexpected behavior.

3.3 Terms

```
⟨term⟩ ::= ⟨termPrimitive⟩
        |  ⟨term⟩ ‘+’ ⟨term⟩ | ⟨term⟩ ‘-’ ⟨term⟩
        |  ⟨term⟩ ‘*’ ⟨term⟩ | ⟨term⟩ ‘/’ ⟨term⟩
        |  ‘!’ ⟨term⟩
        |  ⟨identifier⟩ ‘(’ ⟨term⟩ * ‘)’
        |  ⟨term⟩ ‘[’ ⟨term⟩ ‘]’
        |  ⟨term⟩ ‘.’ ⟨identifier⟩
        |  ‘(’ ⟨type⟩ ‘)’ ⟨term⟩
```

$$\frac{t_1 : T_1, \cdots, t_n, T_n, f : T_1 \to \cdots T_n \to T}{f(t_1, \cdots, t_n) : T} \text{ T-Call}$$

$$\frac{t_{arr} : T\ [n], 0 \leq i < n}{t_{arr}[i] : T} \text{ T-Array}$$

$$\frac{t_{slice} : T\ [\,], 0 \leq i < len(t_{slice})}{t_{slice}[i] : T} \text{ T-Slice}$$

$$\frac{t_{map} : \text{map } [T_1]\ T_2, t_{key} : T_1}{t_{map}[t_{key}] : T_2} \text{ T-Map}$$

$$\frac{t_{struct} : \text{struct } \{field_1 : T_1, \cdots, field_n : T_n\}, 1 \leq i \leq n}{t_{struct}.field_i : T_i} \text{ T-struct}$$

Rules of operators (e.g. $+, -, *$) are not presented here. That's mainly due to their abstract nature. In MediatE, native semantics of numeric operators is absent on syntax level. In other words, we have to manually implement these operators as functions, for example.

Example 1 (Operator Overriding).

```
native function operatorAdd (a:int,b:int):int;
native function <l1:int,l2:int,u1:int,u2:int> operatorAdd(a:int l1 ..
     u1, b:int l2 .. u2) : int (l1 + l2) .. (u1 + u2);
```

3.4 Channels, Connectors and Components

In this subsection, we introduce several basic functional blocks in MediatE, they are : *channels*, the atomic coordination units; *components*, the atomic functional units; and their mixture *connectors*.

All these terms are quite popular concepts in component-based software engineering. In a common senario, components are developed and tested separately by different developers (teams), then collected and organized by connectors which define how these components interact with each other. Basically, complex connectors are composed of simpler ones, among which the primitive ones are called channels.

Definition 3 (Channel). *Channels are the atomic functional units in our language. A channel compries several parameters; a set of ports, either* input *or* output*; some private variables; and a series of* ordered *transitions.*

$\langle channel \rangle$ ::= '**channel**' [$\langle template \rangle$] $\langle identifier \rangle$ ($\langle port \rangle^{*}$)

$$\begin{array}{rcl}
 & & \text{`}\mathbf{\{}\text{'}\ \langle \mathit{variables} \rangle\ \langle \mathit{transitions} \rangle\ \text{`}\mathbf{\}}\text{'} \\
\langle \mathit{port} \rangle & ::= & \langle \mathit{identifier} \rangle\ \text{`}\mathbf{:}\text{'}\ (\ \text{`}\mathbf{in}\text{'} \mid \text{`}\mathbf{out}\text{'}\ )\ \langle \mathit{type} \rangle \\
\langle \mathit{template} \rangle & ::= & \text{`}\langle\text{'}\ \langle \mathit{param} \rangle^{+}\ \text{`}\rangle\text{'} \\
\langle \mathit{param} \rangle & ::= & (\ \text{`}\mathbf{type}\text{'} \mid \langle \mathit{type} \rangle\ )\ \langle \mathit{identifier} \rangle \\
\langle \mathit{variables} \rangle & ::= & \text{`}\mathbf{variables}\text{'}\ \text{`}\mathbf{\{}\text{'}\ (\ \langle \mathit{identifier} \rangle\ \langle \mathit{type} \rangle\ [\ \text{`}\mathbf{init}\text{'}\ \langle \mathit{term} \rangle\ ])^{*}\ \text{`}\mathbf{\}}\text{'}
\end{array}$$

Essentially, behavior of a channel is encoded as a set of *guarded transitions*.

Before introducing the formal grammar of channels, we present a simple example here.

```
channel <T:type> Sync (A: in T, B: out T) {
    transitions {
        _ -> A.reqRead := B.reqRead ;
        _ -> B.reqWrite := A.reqWrite ;

        (A.reqWrite && B.reqRead) -> {
            perform A;
            B.value := A.value;
            perform B;
        }
    }
}
```

Fig. 1. A Simple Synchronous Channle in MediatE

Template. A channel may comes up with a template, which is a set of constant values or types (we denote them as *parameters* hereinafter) that is specified while being instantiated. Templates make it able to create a family of similar channel instances with only a single definition. Parameters in a template can be refered in all the rest declarations, including the interface and variables section.

Interface. Interfaces describe through what ports the channels communicate with its environment. As shown in the example above, `A:in T, B:out T` is the interface of a Synchronous channel, indicating that this channel have an input port and an output port, both of type A, which is provided as a template parameter.

Variables. Two types of variables may show up in a channel's definition, they are:

1. Local Variables.
2. A special family of variables, called *adjoint variables*, are used to describe the status of ports. For a port A, we assume that it has two boolean fields `A.reqRead` and `A.reqWrite` indicating if there is a pending *read* or *write* request on this port, and a data field `A.value` indicating the current value of this port (if a write operation is performed, `A.value` will be reassigned).

A resonable rule comes up that, both the `reqWrite` field of a input port and the `reqRead` field of a output port are *read-only*. Similarly, we cannot rewrite the `value` field of a input port.

Transitions. A transition is a guarded command that is executed when

$$
\begin{aligned}
\langle \textit{transitions} \rangle &::= \text{`}\textbf{transitions}\text{' `}\{\text{' } (\ \langle \textit{transition} \rangle \mid \langle \textit{group} \rangle\ )^{*} \text{ `}\}\text{'} \\
\langle \textit{transition} \rangle &::= \langle \textit{term} \rangle \ \rightarrow \\
&\quad\ (\ \langle \textit{statement} \rangle \mid \text{`}\textbf{begin}\text{' } \langle \textit{statement} \rangle^{+} \text{ `}\textbf{end}\text{' }) \\
\langle \textit{statement} \rangle &::= \langle \textit{identifier} \rangle^{+} \text{ `}\mathbf{:=}\text{' } \langle \textit{term} \rangle^{+} \\
&\ \mid \ \text{`}\textbf{perform}\text{' } \langle \textit{identifier} \rangle^{+} \\
\langle \textit{group} \rangle &::= \text{`}\textbf{group}\text{' `}\{\text{' } \langle \textit{transition} \rangle^{+} \text{ `}\}\text{'}
\end{aligned}
$$

1. *Urgent.*
2. *Ordered.*

3.5 Connectors

$$
\begin{aligned}
\langle \textit{connector} \rangle &::= \text{`}\textbf{connector}\text{' } [\ \langle \textit{params} \rangle\ ]\ \langle \textit{identifier} \rangle\ \mathbf{(}\ \langle \textit{ports} \rangle\ \mathbf{)} \\
&\quad\ \text{`}\{\text{' } [\ \langle \textit{internals} \rangle\ ](\ \langle \textit{connection} \rangle\ )^{+} \text{ `}\}\text{'} \\
\langle \textit{interals} \rangle &::= \text{`}\textbf{internal}\text{' } (\ \langle \textit{identifier} \rangle\ )^{+} \\
\langle \textit{connection} \rangle &::= \langle \textit{identifier} \rangle \text{ `}\mathbf{(}\text{' } \langle \textit{identifier} \rangle^{+} \text{ `}\mathbf{)}\text{'}
\end{aligned}
$$

Example 2 (Alternator in Reo).

```
connector <T:type> alternator (A:in T, B:in T, C: out T) {
    internals A1,A2,B1,B2,C1,C2;
    connections {
        reo.original.Replicator<T>(A,A1,A2);
        reo.original.Replicator<T>(B,B1,B2);
        reo.original.Merger<T>(C1,C2,C);

        reo.original.Sync<T>(A1,C1);
        reo.original.SyncDrain<T>(A2,B1);
        reo.original.Fifo1<T>(B2,C2);
    }
}
```

4 Semantics

5 Discussion

6 Conclusion

[1]

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
2. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of CAV 2011. LNCS, vol. 6806, pp. 1–6. Springer (2011)