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.

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

3.1 Type System

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

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

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 an empty value is strongly required to denote uninitialized value. 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.*

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

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

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

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

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

Definition 3 (Abstract and Concrete Types). *The types which have default values specified are called concrete types. And on contrary, those without default values are termed abstract types.*

$$\langle concreteType \rangle \ ::= \ \langle extendedType \rangle \text{ '}\textbf{init}\text{' } \langle term \rangle$$

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.

3.2 Terms

$$\begin{aligned}
\langle term \rangle \ ::= \ & \langle termPrimitive \rangle \\
| \ & \langle term \rangle \text{ '}+\text{' } \langle term \rangle \mid \langle term \rangle \text{ '}\textbf{-}\text{' } \langle term \rangle \\
| \ & \langle term \rangle \text{ '}\textbf{*}\text{' } \langle term \rangle \mid \langle term \rangle \text{ '}/\text{' } \langle term \rangle \\
| \ & \text{'}\textbf{!}\text{' } \langle term \rangle \\
| \ & \langle identifier \rangle \text{ '}\textbf{(}\text{' } \langle term \rangle^* \text{ '}\textbf{)}\text{'} \\
| \ & \langle term \rangle \text{ '}[\text{' } \langle term \rangle \text{ '}]\text{'} \\
| \ & \langle term \rangle \text{ '}\textbf{.}\text{' } \langle identifier \rangle
\end{aligned}$$

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

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

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

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

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

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

3.3 Channels and Components

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.

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.

$$\begin{aligned}
\langle channel \rangle\ ::=\ & \text{`}\mathbf{channel}\text{'}\ [\ \langle template \rangle\ ]\ \langle identifier \rangle\ (\ \langle port \rangle\ ^*) \\
& \text{`}\mathbf{\{}\text{'}\ \langle variables \rangle\ \langle transitions \rangle\ \text{`}\mathbf{\}}\text{'} \\
\langle port \rangle\ ::=\ & \langle identifier \rangle\ \text{`}\mathbf{:}\text{'}\ (\ \text{`}\mathbf{in}\text{'}\ |\ \text{`}\mathbf{out}\text{'}\ )\ \langle type \rangle
\end{aligned}$$

Templates make it able to create a family of similar channels with a single definition. That is, we are able to declare a set of parameters in the channel's definition. And when creating channel instances, you have to specify concrete

values to these parameters. A parameter here can be either a type identifier (decorated with prefix **type**), or a normal variable.

$$\begin{aligned} \langle template \rangle &::= \text{`}\langle\text{'}\ \langle param \rangle^{+}\ \text{`}\rangle\text{'} \\ \langle param \rangle &::= (\ \textbf{`type'} \mid \langle type \rangle\ )\ \langle identifier \rangle \end{aligned}$$

Variables. Local variables can be declared in the *variables* segement, and referenced in the *trnasitions* segement.

$$\langle variables \rangle ::= \textbf{`variables'}\ \textbf{`\{'}\ (\ \langle identifier \rangle\ \langle type \rangle\ [\ \textbf{`init'}\ \langle term \rangle\ ])^{*}\ \textbf{`\}'}$$

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

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

3.4 Connectors

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

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)