Palang Toy Language: Formal Discription

Ramtin Khosravi

1 Introduction

Palang is a toy actor modeling language used to study various aspects of actor systems. An example of a Palang model is illustrated in Figure 1. Palang is based on Rebeca: Reactive Objects Langauge, an actor based modeling language, but has omitted a number of features to make it simpler. There is no notion of class in Palang, and each actor is described individually. So, in the main block, only the messages that are sent to initiate the computation are specified. An actor may send messages to any other actor, hence, there is no 'knownrebecs' part. Nondeterministic assignment is omitted from the set of statements, and a special 'skip' statement is introduced (that does nothing). The 'initial' messages no longer exist. It is assumed that all the state variables are initialized to zero. If needed, an ordinary message must be used to initialize the variables.

A few syntactic changes are made to Rebeca too. The 'statevars' block no longer exists and the state variables may be defined anywhere within the actor's declaration. The keyword 'msgsrv' is also removed, and the message servers are referred to as 'methods'. Finally, the message send operator has been changed from '.' to '!' to give a more sense of asynchrony. The formal syntax of Palang is described in EBNF notation in Figure 2 (at the end of the article).

In this article, we describe the formal operational semantics of Palang using transition systems. Two types of semantics will be defined for Palang based on fine- or coarse-grained execution of the methods.

2 Notation

In this article, we have tried to use standard notation wherever possible. This part reviews less standard notations used in the rest of the article.

We use the following notations for working with sequences. Given a set A, the set A^* is the set of all finite sequences over elements of A. For a sequence $a \in A^*$ of length n, the symbol a_i denotes the i^{th} element of the sequence, where $1 \leq i \leq n$. In this case, we may also write a as $\langle a_1, a_2, \ldots, a_n \rangle$. The empty sequence is represented by ϵ, and $\langle h|T \rangle$ denotes a sequence whose first elements is $h \in A$ and $T \in A^*$ is the sequence comprising the elements in the rest of the sequence. For two sequences σ and σ' over A, $\sigma \oplus \sigma'$ is the sequence obtained by appending σ' to the end of σ.

```
1  actor a {
2    int x;
3
4    n() {
5      x = x + 1;
6      if (x < 2)
7        b!m(x);
8      else
9        self!n();
10   }
11 }
```

```
13 actor b {
14   int i;
15
16   m(int j) {
17     i = j + 1;
18     a!n();
19   }
20 }
21
22 main {
23   b!m(3);
24 }
```

Figure 1: An example Palang model

For a function $f : X \to Y$, we use the notation $f[x \mapsto y]$ to denote the function $\{(a, b) \in f | a \neq x\} \cup \{(x, y)\}$. We also use the notation $x \mapsto y$ as an alternative to (x, y). For $X' \subseteq X$, we write $f|_{X'}$ as the restriction of f to X', i.e., $\{(x, y) \in f | x \in X'\}$. Having two sequences a and b of the same size n, the function $map(a, b)$ returns the mapping of the elements of a into b such that $map(a, b) = \{a_i \mapsto b_i | 1 \leq i \leq n\}$, assuming that the elements of a are distinct.

3 Abstract Syntax

To enable formal description of Palang semantics, we first provide an abstract specification of a Palang model's syntax. A Palang model consists of a number of actor declarations and a `main` block specifying initial messages to the actors.

3.1 Actors

We assume each actor is an instance of the type $Actor = ID \times 2^{Var} \times 2^{Mtd}$, where:

- ID is the set of all actor identifiers in the model
- Var is the set of all variable names
- Mtd is the set of all method declarations

An actor $(id, vars, mtds)$ has the identifier id, the set of state variables $vars$, and the set of methods $mtds$. Each method is defined as the triple $(m, p, b) \in MName \times Var^* \times Stat^*$, where m is the name of the message the method is used to serve, p is the sequence of the names of the formal parameters, and b contains the sequence of statements comprising the body of the method.

3.2 Statements

The set of statements is defined as $Stat = Assign \cup Cond \cup Send \cup \{\mathsf{skip}\}$, where different types of statements are defined as below.

- $Assign = Var \times Expr$ is the set of assignment statements. We use the notation $var := expr$ as an alternative to $(var, expr)$.
- $Cond = BExpr \times Stat^* \times Stat^*$ is the set of conditional statements. We use the notation $\mathsf{if}\ expr\ \mathsf{then}\ \sigma\ \mathsf{else}\ \sigma'$ as an alternative to $(expr, \sigma, \sigma')$.
- $Send = (ID \cup \{self\}) \times MName \times Expr^*$ is the set of send statements. We use the notation $x!m(e)$ as alternative to (x, m, e).
- skip is a predefined statement that has no effect.

The meaning of the above statements is straightforward. *Expr* denotes the set of integer expressions defined over usual arithmetic operators (with no side effects). *BExpr* denotes the set of Boolean expressions defined over usual relational and logic operators. We do not dig into the details of the expressions in this report.

3.3 Main Block

Having the above definitions, the set of Palang models is specified by $2^{Actor} \cup Send^*$, where the second component corresponds to the main block consisting of a sequence of message send statements. Note that since there may be more than one message to the same actor, the send statements are ordered in a sequence and not just a set of statements.

3.4 Auxiliary Functions

We define the following auxiliary functions to be used in defining the formal semantics:

- $body : ID \times MName \rightarrow Stat^*$, in which $body(x, m)$ returns the body of the method m of the actor identified by x, appended by the special element endm, which denotes the end of the method.
- $params : ID \times MName \rightarrow Var^*$, in which $params(x, m)$ returns the list of formal parameters of the method m of the actor identified by x.
- $svars : ID \rightarrow 2^{Var}$ which returns the names of the state variables of the actor identified by x.

3.5 Static Semantics

The following rules define the well-formedness of a Palang model which is hard to (or cannot be) described in the Palang grammar, but may be statically checked.

Unique Identifiers. The actor identifiers are unique within a Palang model.

Unique Variables. The names of the state variables of an actor are unique.

Unique Methods. The names of the methods of an actor are unique.

Unique Parameters. The names of the formal parameters of a method are unique and different from the state variables of the enclosing actor.

Type Safety. The model is well typed, i.e.,

- the expressions are well-typed,
- both sides of an assignment are of the same type,
- the conditions of the conditional statements are of type Boolean, and
- the receiver of a message has a method with the same name as the message.

Well-Formed Arguments. The list of actual arguments passed to a message send statement conforms to the list of formal parameters of the corresponding method, in both length and type.

4 Operational Semantics

In this section, we describe the formal semantics of Palang in terms of transition systems. But before that, we make a few definitions and assumptions.

We assume the set Val contains all possible values that can be assigned to the state variables or to be used within the expressions. Here, we have $Val = \mathbb{Z} \cup \{\text{True}, \text{False}\}$. As the main focus is on the message passing and interleavings of actor's execution, we abstract away the semantics of expressions by assuming the function $eval_v : Expr \to Val$ evaluates an expression within the specific context $v : Var \to Val$. We assume $eval_v$ is overloaded to evaluate a sequence of expressions: $eval_v(\langle e_1, e_2, \ldots, e_n \rangle) = \langle eval_v(e_1), eval_v(e_2), \ldots, eval_v(e_n) \rangle$.

4.1 States

We assume actors communicate via message passing and queue their incoming messages in a FIFO mailbox. We define the type for the messages as $Msg = MName \times (Var \to Val)$. In a message $(m, a) \in Msg$, m is the name of the message and a is a function mapping argument names to their values. The mailbox of an actor is defined as a sequence of messages, written as Msg^*.

The global state of a Palang system is represented by a function $s : ID \to (Var \to Val) \times Msg^* \times Stat^*$, which maps an actor's identifier to the local state of the actor. The local state of an actor is defined by a triple like (v, q, σ), where $v : Var \to Val$ gives the values of the state variables of the actor, $q : Msg^*$ is the mailbox of the actor, and $\sigma : Stat^*$ contains the sequence of statements the actor is going to execute to finish the service to the message currently being processed.

4.2 Transitions

Here, we define the transitions between states that occur as the results of actors' activities including: taking a message from the mailbox, executing statements, and ending the execution of a method. The following SOS rules define these transitions.

$$\frac{s(x) = (v, \langle (m, a) | T \rangle, \epsilon)}{s \to s[x \mapsto (v \cup a \cup \{(self, x)\}, T, body(x, m))]} \quad \text{(message take)}$$

$$\frac{s(x) = (v, q, \langle var := expr | \sigma \rangle)}{s \to s[x \mapsto (v[var \mapsto eval_v(expr)], q, \sigma)]} \quad \text{(assignment)}$$

$$\frac{s(x) = (v, q, \langle \mathsf{if}\ expr\ \mathsf{then}\ \sigma\ \mathsf{else}\ \sigma' | \sigma'' \rangle) \wedge eval_v(expr) = \text{True}}{s \to s[x \mapsto (v, q, \sigma \oplus \sigma'')]} \quad \text{(conditional}_T\text{)}$$

$$\frac{s(x) = (v, q, \langle \mathsf{if}\ expr\ \mathsf{then}\ \sigma\ \mathsf{else}\ \sigma' | \sigma'' \rangle) \wedge eval_v(expr) = \text{False}}{s \to s[x \mapsto (v, q, \sigma' \oplus \sigma'')]} \quad \text{(conditional}_F\text{)}$$

$$\frac{s(x) = (v, q, \langle y!m(e) | \sigma \rangle) \wedge s(y) = (v', q', \sigma') \wedge p = params(y, m)}{s \to s[x \mapsto (v, q, \sigma)][y \mapsto (v', q' \oplus \langle (m, map(p, eval_v(e))) \rangle, \sigma')]} \quad \text{(send)}$$

$$\frac{s(x) = (v, q, \langle \mathsf{skip} | \sigma \rangle)}{s \to s[x \mapsto (v, q, \sigma)]} \quad \text{(skip)}$$

$$\frac{s(x) = (v, q, \langle \mathsf{endm} \rangle)}{s \to s[x \mapsto (v|_{svars(x)}, q, \epsilon)]} \quad \text{(end-of-method)}$$

Now we can define the transition system semantics of a Palang model $\mathcal{P}$ as the triple $TS(\mathcal{P}) = (S, \to, s_0)$, where:

- S is the set of global states (i.e., the set of all functions from actor identifiers to local states),
- $\to$ is the smallest relation defined by the above SOS rules, and
- s_0 is the initial state as defined below.

In the initial state, all state variables have zero values and the messages specified in the main block are put in the corresponding actors' mailboxes: $s_0(x) = (\{var \mapsto 0 | var \in svars(x)\}, q(x, \sigma), \epsilon)$, where $q(x, \sigma)$ constructs the initial mailbox of actor x from the sequence of send statements σ (from the main block) as defined below:

$$\begin{aligned} q(x, \epsilon) &= \epsilon \\ q(x, \langle x!m(e) | \sigma' \rangle) &= \langle (m, map(p, eval_\emptyset(e))) | q(x, \sigma') \rangle \\ q(x, \langle y!m(e) | \sigma' \rangle) &= q(x, \sigma') \end{aligned}$$

In the above definition, it is assumed that $x \neq y$ and $p = params(x, m)$. Note that since there are no local variables defined in the main block, the context for evaluation of the arguments of the messages in this block is the empty mapping (hence, $eval_\emptyset$ is used).

4.3 Coarse-Grained Semantics

In Rebeca, the execution of the message servers (methods) are non-preemptive, i.e., when an actor takes a message, it executes the entire body of the message server before starting execution of another message server. We call this type of semantics *coarse-grained as opposed to our semantics description of Palang which is* fine-grained. To define a coarse-grained semantics of a Palang model $\mathcal{P}$, we define the transition relation $\Rightarrow \subset S^2$.

An actor in the local state (v, q, ϵ) is called *idle*, i.e., it is not in the middle of processing a message. A global state s is idle, if $s(x)$ is idle for every actor identifier x in the model. We use the notation $idle(s, x)$ to denote the actor identified by x is idle in state s, and $idle(s)$ to denote all actors in s are idle.

Two global states s and s' are in relation $\Rightarrow$ iff both are idle, and there is a path between s and s' in $TS(\mathcal{P})$ such that the first transition on the path is caused by an actor x "taking" a message (written as $\xrightarrow{t_x}$), and every other actor is idle throughout the entire path. Note that all outgoing transitions from an idle state are "message take" transitions. Formally, $s \Rightarrow s'$ iff

- $idle(s) \wedge idle(s')$, and
- $\exists s_1, s_2, \ldots s_k \in S, x \in ID \cdot s = s_1 \xrightarrow{t_x} s_2 \rightarrow \ldots \rightarrow s_k = s' \wedge \forall y \in ID, 1 \leq j \leq k \cdot y = x \vee idle(s_j, y)$.

This way, we can define the coarse-grained transition system $CTS(\mathcal{P}) = (\{s \in S | idle(s)\}, \Rightarrow, s_0)$. Note that s_0 is defined as the same as the fine-grained transition system.

4.4 Labeling the Transition System

When defining the transition system semantics of Palang, we did not label the transitions with *actions*, nor did we label the states with *atomic propositions*. Depending on the purpose the semantics is to be used, proper labels may be attached to the states and/or transitions.

⟨*model*⟩ ::= ⟨*actor*⟩* ⟨*main*⟩

⟨*actor*⟩ ::= '`actor`' ⟨*actor-id*⟩ '`{`' (⟨*state-var*⟩ | ⟨*method*⟩)* '`}`'

⟨*state-var*⟩ ::= ⟨*var-decl*⟩ '`;`'

⟨*var-decl*⟩ ::= ⟨*type*⟩ ⟨*var*⟩

⟨*method*⟩ ::= ⟨*message*⟩ '`(`' ⟨*arg-list*⟩ '`)`' '`{`' ⟨*stat-list*⟩ '`}`'

⟨*arg-list*⟩ ::= ϵ | ⟨*var-decl*⟩ ('`,`' ⟨*var-decl*⟩)*

⟨*stat-list*⟩ ::= (⟨*statement*⟩ '`;`')*

⟨*statement*⟩ ::= ⟨*assignment*⟩ | ⟨*conditional*⟩ | ⟨*send*⟩ | ⟨*skip*⟩

⟨*assignment*⟩ ::= ⟨*var*⟩ := ⟨*expr*⟩

⟨*conditional*⟩ ::= '`if`' '`(`' ⟨*expr*⟩ '`)`' ⟨*stat-list*⟩ '`else`' ⟨*stat-list*⟩

⟨*send*⟩ ::= ⟨*actor-id*⟩ '`!`' ⟨*message*⟩ '`(`' ⟨*expr-list*⟩ '`)`'

⟨*skip*⟩ ::= '`skip`'

⟨*expr-list*⟩ ::= ϵ | ⟨*expr*⟩ ('`,`' ⟨*expr*⟩)*

⟨*expr*⟩ ::= *expressions over usual (side-effect free) operators*

⟨*main*⟩ ::= '`main`' '`{`' (⟨*send*⟩ '`;`')* '`}`''

⟨*message*⟩ ::= ⟨*identifier*⟩

⟨*actor-id*⟩ ::= ⟨*identifier*⟩

⟨*var*⟩ ::= ⟨*identifier*⟩

⟨*type*⟩ ::= '`int`'

Figure 2: The grammar of Palang in EBNF – the detailed syntax for expressions is omitted