Muds$^{(\mathrm{LTS})}$ Syntax and Semantics

(20.11.2018 12:52)

For reference, we provide the formal syntax and semantics of the MUDS language. MUDS is designed to be a toy example of a guarded command language as seen and used in the lecture "Verification" and its project (until mid January 2019). MUDS has been designed as a fragment of the input language of the MODEST model checker[1]. As a consequence, any MUDS program can be used as an input for the latter model checker to check, although the converse is not true (MODEST provide more modelling features).

1 Variables and Expressions

In this section, we give abstract definitions of several concepts related to variable manipulation and in particular introduce different classes of expressions that will be used in different places.

1.1 Variables and Valuations

A *variable* x is an object that has an associated domain (or *type*) $\mathrm{Dom}(x)$. We typically use the symbol *Var* to denote (finite) sets of variables. Common types of variables include Boolean variables, where $\mathrm{Dom}(x) = \mathbb{B} = \{\mathit{true}, \mathit{false}\} = \{\mathit{tt}, \mathit{ff}\}$; integer variables, where $\mathrm{Dom}(x) = \mathbb{Z}$.

For a set of variables *Var*, we let $\mathrm{Val}(\mathit{Var})$ denote the set of variable *valuations*, that is of functions $\mathit{Var} \to \bigcup_{x \in \mathit{Var}} \mathrm{Dom}(x)$ that map variables to values such that $v \in \mathrm{Val}(\mathit{Var}) \Rightarrow \forall\, x \in \mathit{Var}\colon v(x) \in \mathrm{Dom}(x)$. When *Var* is clear from the context, we may write *Val* in place of $\mathrm{Val}(\mathit{Var})$. If *Var* is finite and we are given an ordering on the variables (or we assume one), we can represent a valuation v as a tuple whose i-th component is the value of the i-th variable according to v. Two valuations $v_1 \in \mathrm{Val}(\mathit{Var}_1)$ and $v_2 \in \mathrm{Val}(\mathit{Var}_2)$ are *consistent* if $x \in \mathit{Var}_1 \cap \mathit{Var}_2 \Rightarrow v_1(x) = v_2(x)$. Then their union $v_1 \cup v_2$ is a valuation in $\mathrm{Val}(\mathit{Var}_1 \cup \mathit{Var}_2)$.

1.2 Expressions

Given a set of variables *Var*, by $\mathrm{Exp}(\mathit{Var})$ we denote the set of *expressions* over the variables in *Var*. When *Var* is clear from the context, we may write *Exp* in place of $\mathrm{Exp}(\mathit{Var})$. We treat the *syntax* of expressions in an abstract manner and formally work mostly with the *semantics* of expressions as functions that take a valuation over the relevant variables and return some kind of value depending on the expression class. For an expression e, we denote its function semantics by $[\![e]\!]$, and consequently write its evaluation for a given valuation v as $[\![e]\!](v)$. Just like a variable has a domain, an expression has a type $t = \mathit{type}(e)$. We denote by $e[x/e']$ the replacement of all occurrences of variable x in the syntax of the expression e by the expression e'. Formally, this means that

$$\forall\, v \in \mathit{Val}\colon [\![e[x/e']]\!](v) = [\![e]\!](v[x \mapsto [\![e']\!](v)]).$$

The functional expression semantics can formally be defined as follows:

Definition 1 (Expression semantics)**.** The semantics $[\![e]\!]$ of an expression e over variables in *Var* with $\mathit{type}(e) = t$ is such that $[\![e]\!] \in \mathit{Val} \to t$.

[1] `https://www.modestchecker.net/`

1.3 Assignments and Consistency

An *assignment*, which we can write as $x := e$, is formally a pair $\langle x, e \rangle$ of a variable x and an expression e. The set of assignments to variables in *Var* is $\mathrm{Asgn}(Var) = Var \times \mathrm{Exp}(Var)$, or just *Asgn* if *Var* is clear from the context, such that if $\langle x, e \rangle \in Asgn$, then for all valuations v we have $[\![e]\!](v) \in \mathrm{Dom}(x)$. Two assignments $\langle x_1, e_1 \rangle$ and $\langle x_2, e_2 \rangle$ are *consistent* if $x_1 \neq x_2$ or $[\![e_1]\!](v) = [\![e_2]\!](v)$ for all valuations v. A finite set of pairwise consistent assignments is called an (atomic) *update*, and two updates are consistent if their union is an update. The set of all updates to variables in *Var* is $\mathrm{Upd}(Var)$, or just *Upd* when *Var* is clear from the context. We can identify the assignment u with the update $\{ u \}$. Due to consistency, we can also treat an update $U = \{ \langle x_1, e_1 \rangle, \ldots, \langle x_n, e_n \rangle \}$ consisting of $n \in \mathbb{N}$ assignments where possibly $x_i = x_j$ for some $i \neq j$ as a function $U \in Var \to Exp$ with

$$U = \{ \langle x, e \rangle \mid (\exists\, i : x = x_i \wedge e = e_i \wedge \nexists\, j < i : x = x_j) \oplus (e = x \wedge \nexists\, i : x = x_i) \}$$

where $\oplus$ denotes the exclusive disjunction operator. This merely means that we ignore assignments that are syntactically different, but semantically equivalent.

The formal semantics of an update U (and thus also of a single assignment u via its corresponding update) can simply be defined as follows:

Definition 2 (Assignment semantics)**.** The semantics $[\![U]\!]$ of an update U is such that $[\![U]\!] \in Val \to Val$, and it is defined by $[\![U]\!](v)(x) \stackrel{\text{def}}{=} [\![U(x)]\!](v)$.

The set of variables of an expression, assignment or update y is

$$\mathrm{Var}(y) = \{ x \in Var \mid \exists\, v \in Val, z \in \mathrm{Dom}(x) : [\![y]\!](v) \neq [\![y]\!](v[x \mapsto z]) \},$$

that is the set of variables that $[\![y]\!]$ depends on. It is usually a subset of the variables that appear in y on the syntactical level.

2 Syntax

We start our discussion by giving the complete grammar for MODEST processes and process behaviours.

2.1 Models, Processes and Declarations

A MODEST model consists of a sequence of *declarations* and a *process behaviour*. Declarations are constructed according to the following grammar:

dcl ::=	`action` *act*; |	(*actions*)
	`property` *prop* = Φ; |	(*properties*)
	type var [= *e*]; |	(*variables*)
	P	(*process*)

where, *act*, *var* and *prop* are identifiers and *type* is a type (see Table 2 for the list of types),$e \in Exp$ of type *type* and *P* is a process behaviour. A property Φ is composed of propositions, that is to say boolean expressions of type $\mathbb{B}$ over the already defined variables, of boolean operations and of temporal and branching modalities. For

Lecture	MUDS syntax	Meaning
$\wedge$	&&	And
$\vee$	||	Or
$\neg$	!	Not
U	U	**U**ntil
$\bigcirc$	X	Ne**X**t
$\exists$	E	Does there **E**xist a path
$\forall$	A	For **A**ll paths
$\Box$	G	**G**lobally (always)
$\Diamond$	F	Eventually (**F**inally)

Table 1: Correspondance between lecture and actual tool property specifications

syntax	type	domain
`bool`	Boolean variables	$\{true, false\}$
`int`	unbounded integers	$\mathbb{Z}$

Table 2: Some variable types in MODEST

pragmatical reasons, the notations of these modalities are different from the lecture material (see Table 1).

A MODEST model can thus be treated as a process refer to a process in the remainder, we mean the model's unnamed top-level process. The declarations of the process defines the following (finite) sets:

- $Act_P \uplus \{\tau, \bot, \flat\}$, the disjoint union of the set of actions, the set of exceptions, and the silent action τ, the error action $\bot$ and the break action $\flat$;
- Var_P, the set of variables

To simplify our definitions w.l.o.g., we assume that any particular action, or variable is declared in at most one place in a given model.

Variables can initially be assigned the value of an expression $e \in Exp$ explicitly; otherwise, they are implicitly initialised to a default value, typically zero. We treat expressions in an abstract manner here, omitting a full grammar.

2.2 Process Behaviours

The process behaviours are constructed according to the following grammar:

$$P ::= act \mid act\ \{= u_1, \ldots, u_k =\} \mid \texttt{stop} \mid \texttt{abort} \mid \texttt{break} \mid P_1; P_2 \mid$$
$$\texttt{when}(e_b)\ P \mid \texttt{alt}\{::P_1 \ldots ::P_k\} \mid \texttt{do}\{::P_1 \ldots ::P_k\} \mid \texttt{par}\{::P_1 \ldots ::P_k\}$$

where for $j \in \{1, \ldots, k\}$, $act \in Act_Q \cup \{\tau\}$, $\{u_1, \ldots, u_k\} \in Upd$, $e_b \in Exp$ with $type(e_b) = \mathbb{B}$, $e_j \in Exp$, $H \subseteq Act_Q$ is a set of observable actions.

2.2.1 Shorthands

Binary tests can also be written using the `if` shorthand that maps back to `alt` as follows:

$$\texttt{if}(e)\ \{\,P_1\,\}\ \texttt{else}\ \{\,P_2\,\} \stackrel{\text{def}}{=} \texttt{alt}\ \{\,::\texttt{when}(e)\ P_1\ ::\texttt{when}(\neg e)\ P_2\}$$

Similarly, conditional loops can be abbreviated using the `while` shorthand:

$$\texttt{while}(e)\ \{\,P_1\,\} \stackrel{\text{def}}{=} \texttt{do}\ \{\,::\texttt{when}(e)\ P_1\ ::\texttt{when}(\neg e)\ \texttt{break}\,\}$$

Finally, if a `do` loop contains only one process behaviour that is to be executed over and over again, the double colon can be omitted:

$$\texttt{do}\ \{\,P_1\,\} \stackrel{\text{def}}{=} \texttt{do}\ \{\,::P_1\,\}$$

3 Symbolic Semantics

In this section, we define the symbolic semantics of a given MODEST process, which is the first of two steps in defining a semantics for MODEST. It consists in transforming the process calculus constructs of a process into a program graph (PG).[2]

In this PG, the parts of the model not related to process calculus-based definitions, such as model variables or assignments, will still be maintained in a symbolic form. In absence of non-tail-recursive process calls, the graph will stay finite, so that it is possible to build it explicitly. The second step of the MODEST semantics is the transformation of this symbolic graph into a concrete, possibly infinite, labelled transition system. **Exercise: Define this second step.**

Definition 3 (MODEST symbolic semantics)**.** The *symbolic semantics* of a MODEST process behaviour P is the PG

$$\langle Loc, Var, Act, \longrightarrow, l_{init}, AP\rangle$$

where
- the initial values of the variables are determined by setting them to the default value of their type and then applying the update $v_0 = v_{\text{decl}}$ where v_{decl} represents the initial values assigned to all variables in Var according to their declarations.
- $l_{init} = P$,
- AP contains relevant expressions in $\text{Exp}(Var)$ of type $\mathbb{B}$ that occur in the model,
- the edge relation $\longrightarrow$ is given by the inference rules presented below, and
- the set Loc of locations is the set of reachable process behaviours according to $\longrightarrow$.

Exercise: Is this all in line with the material presented in the lecture, or are there differences?

Every edge $\xrightarrow{g,a,U}$ has three labels: the guard g, the action a, and an update U. It leads to a target process behaviour.

3.1 Inference Rules

The edge relation $\longrightarrow$ is given by the following inference rules:

[2] In full MODEST generality, the PG constructed in this step is a stochastic timed automaton (STA) with invariants and deadlines. We will use STA as a synonym for PG in the sequel.

Actions and the like The inference rules for performing an action, including the special action $\flat$ to break out of a loop with the `break` construct, are straightforward:

$$\frac{}{act \xrightarrow{tt,act,\emptyset} \checkmark}\ (act) \qquad \frac{}{act\ \{=u_1,\ldots,u_k=\} \xrightarrow{tt,act,\{u_1,\ldots,u_k\}} \checkmark}\ (act\text{-}assgn)$$

$$\frac{}{\texttt{break} \xrightarrow{tt,\flat,\emptyset} \checkmark}\ (break) \qquad \frac{}{\texttt{abort} \xrightarrow{tt,\bot,\emptyset} \texttt{abort}}\ (abort)$$

The *successfully terminated process* $\checkmark$ is only used as part of the semantics and cannot be specified syntactically. The `abort` process, simply performs the unhandled error action $\bot$ over and over again. There is no inference rule for the `stop` process since its semantics is precisely to do nothing.

Conditions Any process behaviour can be decorated with a guard using the `when` construct. Guards only affect the first, immediate edges resulting from the decorated behaviour and then disappear.

$$\frac{P \xrightarrow{g,a,U} P'}{\texttt{when}(e)\ P \xrightarrow{g \wedge e,a,U} P'}\ (when)$$

Sequential composition A process behaviour Q can be performed only after another process behaviour P has successfully terminated when they are composed using the ; operator for sequential composition:

$$\frac{P \xrightarrow{g,a,U} P' \qquad P' \neq \checkmark}{P;Q \xrightarrow{g,a,U} P';Q}\ (seq) \qquad \frac{P \xrightarrow{g,a,U} \checkmark}{P;Q \xrightarrow{g,a,U} Q}\ (seq\checkmark)$$

Nondeterministic choice A nondeterministic choice between several process behaviours is provided by the `alt` keyword:

$$\frac{P_i \xrightarrow{g,a,U} P_i \quad (i \in \{1,\ldots,k\})}{\texttt{alt}\ \{::P_1 \ldots ::P_k\} \xrightarrow{g,a,U} P_i}\ (alt)$$

Loops The semantics of the `do` construct is defined via the auxiliary `auxdo` construct, which is not part of the MODEST syntax. It is used to keep track of the original behaviour of the loop which must be restored after each iteration:

$$\texttt{do}\ \{::P_1 \ldots ::P_k\} \stackrel{\text{def}}{=} \texttt{auxdo}\ \{\ \texttt{alt}\ \{::P_1 \ldots ::P_k\}\ \}\ \{\ \texttt{alt}\ \{::P_1 \ldots ::P_k\}\ \}$$

The semantics of `auxdo` is defined in three inference rules: The first one handles the case that the break action is performed to jump out of the loop, while the other rules define the semantics of performing a step within the loop, including proceeding to the next iteration once the last step has been performed:

$$\frac{P \xrightarrow{g,\flat,U} P'}{\texttt{auxdo}\ \{\ P\ \}\ \{\ Q\ \} \xrightarrow{g,\tau,\emptyset} \checkmark}\ (breakout)$$

$\alpha(P) = \{ act \} \setminus \{ \tau \}$	if P has the form act or $act\ \{= u_1, \ldots, u_k =\}$
$\alpha(P) = \varnothing$	if P has one of the following forms: `stop`, `break`, `abort` or `throw`($excp$)
$\alpha(P) = \alpha(Q)$	if P has the form `when`(e) Q
$\alpha(P) = \alpha(P_1) \cup \alpha(P_2)$	if P is of the form $P_1 ; P_2$
$\alpha(P) = \bigcup_{i=1}^{k} \alpha(P_i)$	if P has one of the following forms: `alt` $\{ :: P_1 \ldots :: P_k \}$, `do` $\{ :: P_1 \ldots :: P_k \}$ or `par` $\{ :: P_1 \ldots :: P_k \}$

Table 3: The alphabet of a process behaviour

$$\frac{P \xrightarrow{g,a,U} P' \qquad P' \neq \checkmark \qquad a \neq \flat}{\texttt{auxdo}\ \{\ P\ \}\ \{\ Q\ \} \xrightarrow{g,a,U} \texttt{auxdo}\ \{\ P'\ \}\ \{\ Q\ \}}\ (auxdo)$$

$$\frac{P \xrightarrow{g,a,U} \checkmark \qquad a \neq \flat}{\texttt{auxdo}\ \{\ P\ \}\ \{\ Q\ \} \xrightarrow{g,a,U} \texttt{auxdo}\ \{\ Q\ \}\ \{\ Q\ \}}\ (auxdo\checkmark)$$

Parallel composition The process behaviours in a `par` construct run concurrently, synchronising on the actions in their common alphabet. The alphabet of a process is computed by function α as defined in Table 3. A parallel composition terminates successfully whenever all its components do so, i.e. $\checkmark \parallel_B \checkmark \stackrel{\text{def}}{=} \checkmark$ for any B.

To define the semantics of parallel composition, we resort to the auxiliary operator $\parallel_B$, with $B \subseteq Act$. The `par` construct is then defined as

$$\texttt{par}\ \{ :: P_1 \ldots :: P_k \} \stackrel{\text{def}}{=} (\ldots((P_1 \parallel_{B_1} P_2) \parallel_{B_2} P_3) \ldots) \parallel_{B_{k-1}} P_k$$

with

$$B_j = (\bigcup_{i=1}^{j} \alpha(P_i)) \cap \alpha(P_{j+1}).$$

The behaviour of $\parallel_B$ is that an action or exception $a \notin B$ can be performed autonomously, i.e., without the cooperation of the other parallel component:

$$\frac{P_1 \xrightarrow{g,a,U} P' \quad (a \notin B)}{P_1 \parallel_B P_2 \xrightarrow{g,a,U} P' \parallel_B P_2}\ (lpar) \qquad \frac{P_2 \xrightarrow{g,a,U} P' \quad (a \notin B)}{P_1 \parallel_B P_2 \xrightarrow{g,a,U} P_1 \parallel_B P'}\ (rpar)$$

while the inference rule for synchronisation reads:

$$\frac{P_1 \xrightarrow{g_1,a,U_1} P_1' \quad P_2 \xrightarrow{g_2,a,U_2} P_2' \qquad (U_1 \text{ and } U_2 \text{ are consistent}) \qquad (a \in B)}{P_1 \parallel_B P_2 \xrightarrow{g_1 \wedge g_2, a, U_1 \cup U_2} P_1' \parallel_B P_2'}\ (sync)$$

Inconsistent updates are considered a modelling error.