This document illustrates the use of *SMAUDE* by giving the small-step semantics of a simple synchronous language with arithmetic expressions. Code snippets of the Maude language are used to illustrate explicitly how the infrastructure in *SMAUDE* is extended. Therefore, some familiarity with Maude's syntax is assumed (see [1] for a reference to the Maude language and system).

Consider a language $\mathcal{S}$ that consists of two kinds of elements: *memory* elements $Mem(m, v)$ and *assignment* elements $m{:=}e$, where m denotes a memory name, v denotes a numerical value, and e denotes an arithmetic expression. Arithmetic expressions are recursively formed using memory names, numerical values, and expressions of the form $e_1 + e_2$, where e_1 and e_2 are arithmetic expressions. In this case, set T consists of all elements having the form $Mem(m, v)$ or $m{:=}v$.

The small-step semantics of $\mathcal{S}$ requires the definition of an evaluation function *eval* that takes as inputs a context Γ, which is a set of elements T, and an arithmetic expression e. It is inductively defined on expressions:

$$eval(\Gamma, e) = \begin{cases} v & \text{if } e \text{ is the numerical value } v, \\ v & \text{if } e \text{ is the memory name } m \text{ and } Mem(m, v) \in \Gamma, \\ v_1 + v_2 & \text{if } e \text{ has the form } e_1 + e_2,\ v_i = eval(\Gamma, e_i) \text{ for } i \in \{1, 2\}. \end{cases}$$

The (parametric) atomic relation $\rightarrow_{\mathcal{S}}$ of the language $\mathcal{S}$ is defined for a context Γ by $A \rightarrow_{\mathcal{S}}^{\Gamma} B$ if and only if $A \subseteq \Gamma$, $A = \{Mem(m, v), m{:=}e\}$, $B = \{Mem(m, u), m{:=}e\}$, and $u = eval(A, e)$, for some memory name m, values v and u, and expression e. The semantic relation of the language is the relation $\rightarrow_{\mathcal{S}}^{\Gamma,s}$ (or $\rightarrow_{\mathcal{S}}^{s}$), where s is the $\prec$-maximal $\rightarrow^{\Gamma}$-strategy, Γ is a ground context, and $\prec_{\mathcal{S}}$ is the empty priority.

Example 1. *If* $\Gamma = \{Mem(x, 3), Mem(y, 4), x{:=}y, y{:=}x\}$*, then:*

$$Mem(x, 3), Mem(y, 4), x{:=}y, y{:=}x \ \rightarrow_{\mathcal{S}}^{\Gamma,s} \ Mem(x, 4), Mem(y, 3), x{:=}y, y{:=}x.$$

Language $\mathcal{S}$ is specified by the Maude system module *SIMPLE*, which includes system module *SMAUDE*, and has the following syntax:

```
mod SIMPLE is
  including SMAUDE .

  eq MODULE-NAME = 'SIMPLE .
  ...
endm
```

Note that constant *MODULE-NAME* is identified with the quoted identifier representing the name of module *SIMPLE*.

Element identifiers include the following constructors with sort *Eid*:

```
  op a    : Nat -> Eid [ctor] .
  ops x y : -> Eid [ctor] .
```

Memory elements use constructors x and y for element identifiers, and assignment elements use constructors $a(\_)$ for element identifiers.

Attribute identifiers include the following constructors with sort *Aid*:

```
ops mem body to : -> Aid [ctor] .
```

Memory elements have attribute *mem* as their only attribute, while assignment elements have attributes *body* and *to* as their only attributes. In the syntax of *SIMPLE*, memory element $Mem(x, v)$ and an assignment element $x{:=}e$ can be represented, respectively, by elements

```
< x | mem : v >   < a(1) | body : e, to : x > .
```

Built-in natural numbers are values of the language and addition corresponds to the built-in one in Maude. These are specified in *SIMPLE* with the following subsort and operation declarations:

```
subsort Nat < Val .
op _+_ : Expr Expr -> Expr [ditto] .
```

Expressions are evaluated equationally by following the definition of *eval*:

```
var  C  : Ctx .    vars I J  : Eid .  vars E E' : Expr .
var  M  : Map .    var  N    : Nat .

eq eval(C,N) = N .
eq eval(( < I | mem : N , M > C), I ) = N .
eq eval(C,E + E') = eval(C,E) + eval(C,E') .
```

Atomic rule r–1 specifies the atomic relation of the language:

```
rl [r-1] :
   < I | mem : N > < J | body : E, to : I >
   =>  < I | mem : eval(E) > .
```

The specification of atomic rules is slightly different to the usual specification of rules in rewriting logic. First, in the lefthand side of an atomic rule, it is sufficient to only mention the attributes involved in the atomic transition. In this case, *SMAUDE* will complete each lefthand side term by automatically adding a variable of sort *Map*, unique for each element, before any matching is performed. Second, in the righthand side of an atomic rule, it is sufficient to only mention the elements and the attributes that can change in the atomic step. In this case, *SMAUDE* updates in the current state *only* the attributes of the elements occurring in the righthand side of the rule, while keeping the other ones intact. So, in atomic rule r–1, the only attribute that can change is attribute *mem* of the memory element. Note also that in the righthand side of r–1, a unary version of function *eval*, without mention of any particular context, is used; *SMAUDE* will automatically extend it to its binary counterpart, for the given context, when computing function *max-strat*.

The context Γ in Example 1, written in the syntax of *SIMPLE*, is

```
< x | mem : 3 >  < y | mem : 4 >
< a(1) | body : y, to : x >  < a(2) | body : x, to : y > .
```

Maude's *search* command can be used to compute, for instance, the one-step synchronous semantic relation of the language in Example 1 from context Γ:

```
Maude> search { Γ } =>1 X:Sys .
search in SIMPLE : { Γ } =>1 X:Sys .
Solution 1 (state 1)
states: 2  rewrites: 514 in 53ms cpu (54ms real) (9655 rewrites/second)
X:Sys --> {< x | mem : 4 > < y | mem : 3 >
            < a(1) | body : y, to : x > < a(2) | body : x, to : y > }
No more solutions.
```

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.