SMEDL Manual

University of Pennsylvania

August 6, 2020

Part I of this manual describes how to write a monitor. It discusses the features and syntax of the SMEDL specification language. Part II is about integrating monitors into your existing code. It describes the API for the generated code and the various transport mechanisms available for asynchronous connections.

Part I.
The SMEDL and A4SMEDL Monitoring Language

A SMEDL monitoring system is designed at two levels: the monitor level and the architecture level. Monitor specifications describe a single monitor, that is, a list of state variables, a list of events that the monitor consumes and emits (known as "imported" and "exported" events), and a set of states, transitions, and actions describing the state machine. Architecture specifications describe how multiple monitors come together to form a monitoring system, including which monitor specifications are involved, how instances of the monitors are parameterized, and how imported and exported events are directed.

Monitor specifications are written in `.smedl` files, and there may be multiple in a monitoring system. Architecture specifications are written in `.a4smedl` files and there is exactly one per monitoring system.

1. SMEDL Specifications

A SMEDL specification describes a single monitor. This is where the actual monitoring logic is defined. At its core, a SMEDL monitor is a set of state machines. Transitions are triggered by events, with optional conditions and associated actions. Monitors may also contain state variables, which the conditions and actions can make use of.

1.1. A Basic Monitor

Listing 1 gives an example of a simple monitor. This guide will break it down piece-by-piece to familiarize you with the basic syntax of SMEDL.

```
1 object Adder;
```

Listing 1: A monitor that keeps a running total

```
object Adder;

state:
  float accumulator = 0;

events:
  imported measurement(float);
  exported sum(float);

scenarios:
  main:
    idle
      -> measurement(val) {
        accumulator = accumulator + val;
        raise sum(accumulator);
      }
      -> idle;
```

This line declares the monitor specification name. It must be present at the top of every monitor specification. The name can be any valid SMEDL identifier. To be specific, an upper or lowercase letter, followed by zero or more additional letters, numbers, or underscores.

This is the only line in the file that stands alone. The rest of the specification is split into three main sections, the start of each marked by a heading keyword.

```
3 state:
4   float accumulator = 0;
```

This is the state variables section. State variables can be thought of as global variables for the monitor. They will be useful for conditions and actions (each of which are discussed later). There are seven types (and one alias) available, listed in Table 1 along with the most similar C type for each. This is the only optional section: If there is no need for state variables, it may be omitted entirely.

```
6 events:
7   imported measurement(float);
8   exported sum(float);
```

This is the event declarations section. Events are the input and output of the monitor. Imported events come from outside the monitor and may trigger transitions and actions. Exported events are generated within the monitor and sent out. (There are also internal events, introduced in subsection 1.2.)

Events can carry parameters with them. These will be useful for conditions or actions, discussed later. The parameters have predefined types, declared with the event.

```
10 scenarios:
```

Table 1: The types available in SMEDL

SMEDL	C
`int`	`int`
`float`	`double`
`double`[a]	`double`
`char`	`char`
`string`	`char *` *or* `char[]`
`pointer`	`void *`
`thread`	`pthread_t`
`opaque`	`SMEDLOpaque`[b]

[a] `double` is an alias of `float` for convenience, not a distinct type.
[b] The `SMEDLOpaque` type is dicussed in

```
11  main:
12    idle
13      -> measurement(val) {
14        accumulator = accumulator + val;
15        raise sum(accumulator);
16      }
17      -> idle;
```

Finally, the scenarios section. This contains the actual monitor logic in the form of state machines. There can be multiple scenarios, each representing one state machine; however, this monitor only has one, named `main`.

Each scenario is defined as a list of transitions. Here, we only have one. From the `idle` state, if there is a `measurement` event, the actions in curly braces are performed, then it proceeds to the `idle` state again.

The starting state for the first transition becomes the starting state for the entire scenario.

Actions have a C-like syntax. There are five types of actions:

<state_var> = <expr> assigns a value to a state variable.

<state_var>++ increments a state variable.

<state_var>-- increments a state variable.

raise <event>(<expr>[, <expr> ...]) raises an exported orr internal event.

<function>(<expr>[, <expr> ...]) calls a helper function (see **??**).

Expressions are C-like as well and may use literals, event parameters, and state variables.

1.2. A More Advanced Monitor

Listing 2 shows a monitor that ensures a light does not illuminate until a button is pressed. It demonstrates some features of SMEDL that were not present in the simple monitor above.

This monitor has two scenarios, `input` and `verify`. These act as separate state machines operating in parallel. Scenarios can share data and interact with each other through state variables and internal events.

```
10  internal check();
```

This is an internal event declaration. Internal events can be raised like exported events and trigger transitions like imported events, but do not leave the monitor. They provide a way for actions in one scenario to send data to or trigger a transition in another scenario.

As a result, a single imported event can cause a chain of internal and exported events. This is known as a *macro step*. There can only be one transition per scenario per macro step. This is a guard against infinite loops.

Note that if a single event triggers a simultaneous transition in multiple scenarios, either scenario is permitted to run first. However, any events

Listing 2: A monitor checking if a light stays off until a button press

```
object WeakUntil;

state:
  int light = 0;
  int button = 0;

events:
  imported light_is(int);
  imported button_is(int);
  internal check();
  exported satisfaction();
  exported violation();

scenarios:
  input:
    idle -> light_is(status) {
        light = status;
        raise check();
      } -> idle;
    idle -> button_is(status) {
        button = status;
        raise check();
      } -> idle;

  verify:
    waiting
      -> check() when (!light && !button)
      -> waiting;
    waiting
      -> check() when (button) {raise satisfaction();}
      -> satisfied;
      else {raise violation();}
      -> violated;
    satisfied -> check() -> satisfied;
    violated -> check() -> violated;
```

raised during a macro step are queued until all events raised before it are finished being handled.

For example, given the scenarios in Listing 3, if `event_a` were raised, the macro step would look like this:

1. The transition in `scn_a` would be triggered.
   a) `event_b` is raised. It is queued for later, since `event_a` is still being handled.
   b) `my_var` is set to `100`.
2. `event_b` triggers transitions in `scn_b` and `scn_bb`. These transitions could happen in either order, but we will do `scn_b` first.
   a) In `scn_b`, `event_d(100)` is raised. It is queued for later.
   b) In `scn_bb`, `event_c()` is raised. It is also queued.
3. `event_d` is popped back off the event queue. We will say this was an exported event, so this is the point where it would be emitted from the monitor.
4. Next is `event_c`, triggering a transition in `scn_c`. `event_e` is raised and queued.
5. Finally, `event_e` is dequeued. `scn_a` also has a transition for this event, but since the macro step is not over, that transition is not taken.

1.3. SMEDL Language Reference

2. A4SMEDL Specifications

2.1. A Basic Architecture

2.2. A More Advanced Architecture

2.3. A4SMEDL Language Reference

Listing 3: Example scenarios to demonstrate ordering within a macro step

```
scenarios:
  scn_a:
    start -> event_a() {
      raise event_b();
      my_var = 100;
    } -> state_a;
    state_a -> event_e() -> start;

  scn_b:
    start -> event_b() {
      raise event_d(my_var);
    } -> start;

  scn_bb:
    start -> event_b() {
      raise event_c();
    } -> start;

  scn_c:
    start -> event_c() {
      raise event_e();
    } -> start;
```

Part II.
The SMEDL Programming Interface

The generated monitoring code is separated into multiple layers, ordered here from lowest to highest:

Monitor contains the actual monitor state machines.

Local wrapper manages instances of a single monitor specification and dynamic instantiation.

Global wrapper manages all local wrappers in a single synchronous set and routes events between them.

Transport handles asynchronous communication between global wrappers and the target system. This layer is only generated when the `-t` option is used with `mgen`.

The monitor (SMEDL) specification corresponds with the monitor level and the architecture (A4SMEDL) specification corresponds with the three layers above it.

Most programs will not need to interact with any layer except the transport layer directly. If not interfacing with SMEDL on a low level, it is safe to skip the sections on the monitor, local wrapper, and global wrapper layers.

3. Common API Elements

Header: `smedl_types.h`

Listing 4: The SMEDLValue and SMEDLType definitons

```
typedef enum {SMEDL_INT, SMEDL_FLOAT, SMEDL_CHAR,
    SMEDL_STRING, SMEDL_POINTER, SMEDL_THREAD,
    SMEDL_OPAQUE, SMEDL_NULL} SMEDLType;

typedef struct {
    SMEDLType t;
    union {
        int i;
        double d;
        char c;
        char *s;
        void *p;
        pthread_t th;
        SMEDLOpaque o;
    } v;
} SMEDLValue;
```

3.1. The SMEDLValue Type

The SMEDLValue type is used throughout the API for event parameters, monitor identities, etc. It is defined in Listing 4. Each SMEDLValue contains a type t and a value v. The type is a member of the SMEDLType enum and the value is a union. See Table 2 for the correspondence between the SMEDL types and union members.

The SMEDL_NULL type is special—it does not correspond with an actual SMEDL type. It is used internally in a couple places, but as far as the API is concerned, there is only one significant use: wildcard parameters. In a list of monitor identities, a SMEDLValue with t set to SMEDL_NULL is understood to be a wildcard. The v member is ignored in such cases.

Table 2: Correspondence between SMEDLType and union in SMEDLValue

SMEDLType	Union member
SMEDL_INT	int i
SMEDL_FLOAT	double d
SMEDL_CHAR	char c
SMEDL_STRING	char *s[a]
SMEDL_POINTER	void *p
SMEDL_THREAD	pthread_t th
SMEDL_OPAQUE	SMEDLOpaque o
SMEDL_NULL	–

[a] Must not be NULL and must be null-terminated.

Listing 5: The SMEDLOpaque struct

```
typedef struct {
    void *data;
    size_t size;
} SMEDLOpaque;
```

3.2. The SMEDLOpaque Type

SMEDL's opaque type is meant to represent arbitrary-length chunks of binary data. In that sense, it is somewhat similar to the string type. The difference is that strings cannot contain null characters, since they are null-terminated. The opaque type can, but that means its size must be stored alongside it explicitly. Thus, we have the SMEDLOpaque struct, defined in Listing 5.

Every SMEDLOpaque struct must point to valid data; the data pointer cannot be NULL. There is one exception to this: if the size is zero, data may be NULL (or any other invalid pointer). This goes for SMEDLOpaques

Listing 6: The `SMEDLCallback` type

```
typedef void (*SMEDLCallback)(SMEDLValue *identities,
    SMEDLValue *params, void *aux);
```

passed to callbacks as well.

Note that copies of opaque data are made during internal processing. This could cause issues if there are any self-referential pointers within.

3.3. The `SMEDLCallback` Type and Event Data

The SMEDL API naturally has several functions that pass event data from one place to another. These functions all accept the same three parameters and do not return anything, so to avoid repetition, these parameters are described here.

`SMEDLValue *identities` is an array of the monitor identities. If there is no monitor (i.e. events from the target system) or the monitor has no identities, this can safely be `NULL`. Additionally, where wildcards are allowed, a `SMEDL_NULL` value will represent one.

`SMEDLValue *params` is an array of the event parameters. If the event has no parameters, this can safely be `NULL`.

`void *aux` is a pointer that is passed through from imported events to exported events untouched. It can be used for provenance data.

The callbacks that SMEDL uses for exported events need to have the same signature. The `SMEDLCallback` type (Listing 6) is a function pointer for exactly that.

3.4. Memory Ownership Conventions

The SMEDL API makes extensive use of arrays and pointers. Generally speaking, the following principle holds:

- The caller is responsible for freeing its own memory. If a heap pointer is passed as a parameter, the callee does not take ownership of it.

This applies to callbacks as well. Callbacks cannot assume pointers will remain valid after they return, as the caller may (and often will) immediately free them.

Exceptions to these rules (if any) will be noted with the particular function.

4. Monitor API

5. Local Wrapper API

6. Global Wrapper API

7. Transport Adapters

7.1. RabbitMQ

7.2. File