VUE²

Version 0.9

September 2010

1 Introduction

VUE^2 or (VUEE) which stands for Virtual Underlay Execution Engine, is an emulator for PicOS applications and their underlying networks. VUE^2 is facilitated by the fact that PicOS is a relatively straightforward descendant of a powerful simulator named SMURPH/SIDE. Thus, the Emulation Engine comprising VUE^2 is built around SMURPH, with a natural mapping of many PicOS operations to their SMURPH counterparts.

This document is a constantly evolving draft of VUE^2 description. As the system is being developed, this description will change. Note that VUE^2 is modified not only in the course of its natural evolution (as a system on its own), but also in response to new features being added to PicOS (which have to be mirrored in VUE^2). At present, VUE^2 captures a significant subset of all PicOS features, including drivers for RF modules, pin/sensor interface, external memories, UARTs, the Nokia LCD display, and more. Most importantly, it provides for an easy expression of network models and their virtual deployment. This means that PicOS praxes[1] can be run under VUE^2 on a multitude of virtual nodes behaving as if they were interconnected via realistic wireless links. Those virtual nodes can interface to real-life OSS agents in exactly the same manner as real nodes would. Thus, in addition to providing insights into the behavior of a PicOS praxis in the real world (performance assessment, network planning, also including power budget), VUE^2 also facilitates the development of various agents that have to be interfaced to that praxis to make the overall system complete.

Until version 0.9, this document included a detailed prescription for manually converting PicOS praxes to a VUE^2-compliant form. Presently, that task is taken care of by PiComp (described in a separate document) which comes with the PICOS package.

This document will not be comprehended without the following accompanying documents by Olsonet Communications: *PicOS, version 3.3* [or higher], *PiComp: the PicOS Compiler, version 0.25* [or higher], *VNETI: Versatile NETwork Interface, version 2.5* [or higher], *SIDE/SMURPH: a Modeling Environment for Reactive Telecommunication Systems, version 3.2* [or higher].

2 Similarities and differences with respect to SMURPH

The striking similarity between PicOS and SMURPH is in the thread model. In both environments, a thread describes a finite state machine with the state transition function specified in terms of event wait operations. The rules for aggregating such operations and waking up the threads based on the occurrence of the awaited events are practically identical. In SMURPH, viewed as a simulator, the awaited events are delivered by abstract objects called *Activity Interpreters*, while in PicOS they are triggered by actual physical phenomena (e.g, a packet reception, a character arrival from the UART, and so on).

The first significant difference between the two systems is in the interpretation of time flow. In SMURH, time is purely virtual, which means that formally nobody cares about the actual execution time of the simulation program, but only about the proper marking of the relevant events with virtual time tags. As in all event-driven simulators, the virtual time tags have nothing to do with real time. For example, a significant amount of calculations may be needed to advance the virtual time by a few microseconds, and no computation at all may be required to bypass several hours of idleness caused by no events in the system model. In the first case, the tiny advancement of the virtual time may take several hours of calculations, in the second case, the system may immediately jump several hours into the future in no time at all.

Consequently, the formal useful semantics of SMURPH and PicOS threads are different. The actual execution time of a SMURPH thread is essentially irrelevant (unless it renders the

[1] PicOS term for “applications.”

model execution too long to wait for the results) and all that matters is the virtual delays separating the artificially triggered events. For example, two threads in SMURPH may be semantically equivalent, even though one of them may exhibit a drastically shorter execution time than the other (due to more careful programming and/or optimization). In PicOS, however, the threads are not (just) models but they run the actual thing. Consequently, the execution time of a thread may directly influence the perceived behavior of the PicOS node.

In this context, the following two assumptions make our endeavor worthwhile:

1. PicOS programs are reactive, i.e., they are practically never CPU bound. In other words, the primary reason why a PicOS thread is making no progress is that it is waiting for a peripheral event rather than the completion of a lengthy calculation.

2. If needed (from the viewpoint of model fidelity), an extensive period of CPU activities can be modeled in SMURPH by appropriately (and explicitly) delaying certain state transitions.[2]

Consequently, in most cases, we can ignore the fact that the execution of a PicOS program takes time at all and only focus on reflecting the accurate behavior of the external events. With this approximation, the job of transforming a PicOS praxis into its VUE[2] model becomes reasonably simple. To further increase the practical value of such a model, SMURPH provides for the so-called *visualization mode* of its execution. In that mode, SMURPH tries to map the virtual time of modeled events to real time, such that the user has an illusion of talking to a real application. This is only possible if the network size and complexity allow the simulator to catch up with the model execution to real time. If not, a suitable slow motion factor can be employed. These issues are described in detail in the SMURPH manual.

While the syntax of PicOS threads is close to that of SMURPH processes, there are some differences. First of all, the language of SMURPH is C++, while PicOS is plain C. Second, a PicOS praxis is a one-node program, while the SMURPH model of that praxis must consist of multiple nodes running the same or possibly different programs. Additionally, the praxis code must be supplemented by the models of all those components of reality that are needed by the praxis to run. This brings about three issues:

1. Transforming the syntax of PicOS programs to that acceptable by SMURPH.

2. Putting multiple nodes, possibly with different PicOS praxes, under the umbrella of a single SMURPH program (model).

3. Modeling the peripheral equipment needed by the praxis; also interfacing and parametrizing such models.

The third part is provided as a library of models whose interiors need not be interesting to the developer. This is to say, the third issue does not overly affect the operation of rendering a PicOS praxis executable under VUE[2], as long as the necessary peripherals have their models in the library. This is because the interface (API) to those models looks **exactly** as the PicOS API to the real equipment. On the other hand, the first two issues come into play when the praxis is transformed/adapted for execution under VUE[2]. Before version 0.9 of VUE[2], the adaptation was done manually. By following a set of mostly mechanical rules, the programmer would tweak the praxis source code and augment it with some extra files, such that the same code was accepted both by PicOS and by VUE[2]. The previous version of this document included a detailed description of that procedure. At present, this task is

[2] We were contemplating adding tools to VUE[2] that would make it possible to specify the timing of state execution time in a PicOS thread. Such a specification could indicate that the amount of CPU time needed to run through the state is not trivial. At this time, the prospective usefulness of such a feature is unclear.

accomplished by PiComp, the PicOS compiler, which comes with the PicOS package and is described in a separate document.

3 System outline

Illustration 1 shows schematically (and somewhat simplistically) the two possible ways of compiling a PicOS praxis. The right path corresponds to the "straightforward" case of turning the praxis into one or more (in the case of a multi-program praxis) flash image files that can be uploaded into physical nodes. The left path outlines the transformation of the praxis into its VUE^2 model.

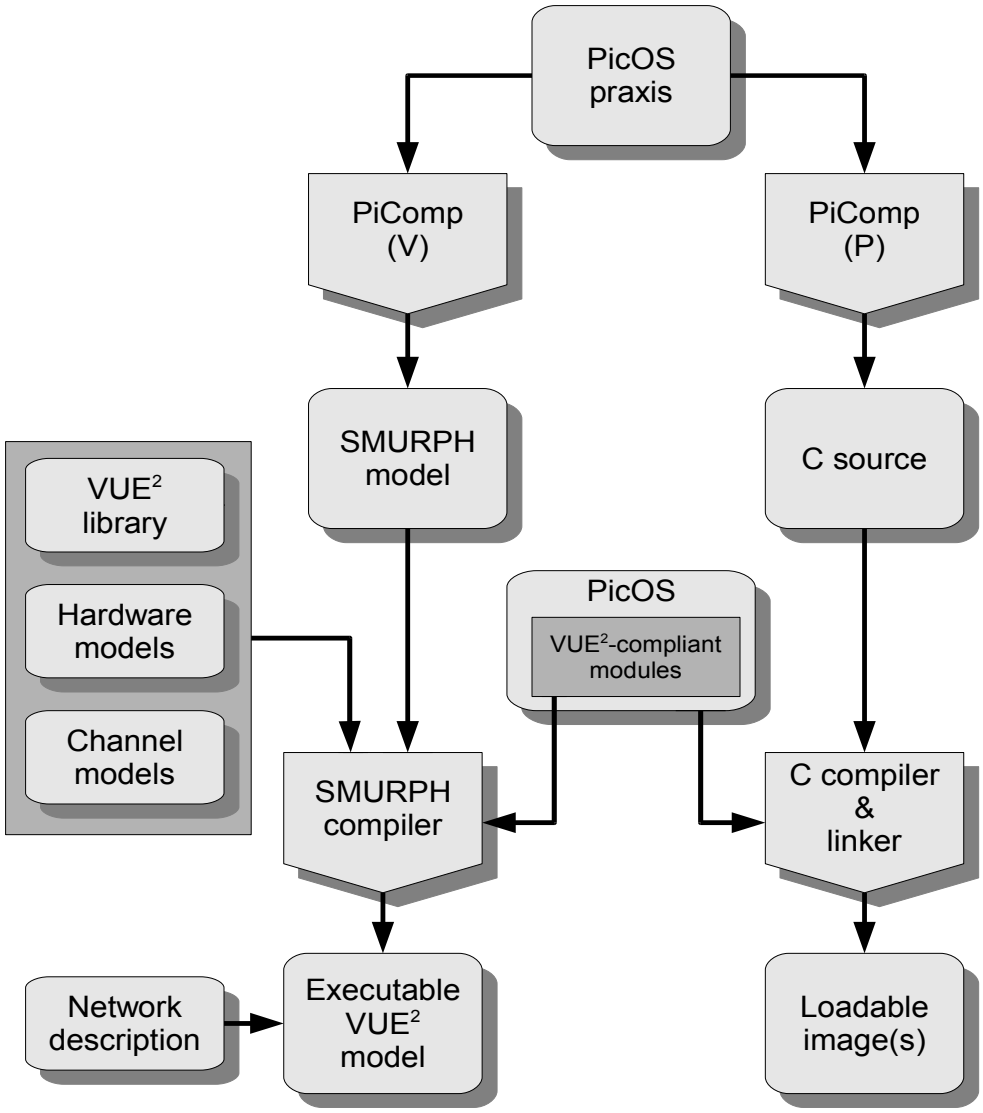

Illustration 1: Compilation paths for a PicOS praxis.

The first stage of either path consists in preprocessing by PiComp (see the *PiComp* document). The exact shape of that preprocessing is different in each case: PiComp is called with different parameters. For the right-hand-side path, the script merely converts PicOS-specific programming constructs to C, such that its output looks like a collection of

programs in plain C that can subsequently be handled by the C compiler. For the VUE^2 path, its task is more elaborate: in addition to translating the PicOS-specific programming constructs, PiComp turns the collection of (possibly multiple) programs comprising the PicOS praxis into a set of source files recognizable by SMURPH/SIDE. This is considerably more difficult. For one thing, it involves encapsulating the node program (or multiple such programs) into a collection of C++ methods that can be executed within the context of a virtual object representing a node, such that multiple virtual nodes can coexist and meaningfully interact within the framework of a complex model. When subsequently processed by the SMURPH compiler, the output of PiComp is turned into a single executable constituting an emulation model for a complete system. That model encompasses a multitude of nodes, wireless channels, and whatever other physical components are needed to fool the praxis into thinking that it operates in a realistic replica of its target environment. Quantitative details of the model's instantiations are described in an input data file to the executable, so it is possible to run it for networks of different sizes, with different distributions of nodes, and so on. It is also possible to interface the virtual nodes to real-life OSS agents, such that those agents can be comfortably developed and tested in a convenient hassle-free setting.

Here is the list of packages contributing to the complete PicOS platform:

1. The standard (vanilla) SMURPH/SIDE package constituting the emulation kernel of the system.

2. PicOS, which in addition to its current collection of VUE^2-compatible praxes provides many files that are directly used by VUE^2 (the VUE2-compliant modules from Illustration 1).

3. VUE^2-specific environment (SMURPH libraries and other files) that in combination with PicOS sources and praxes contribute to VUE^2 models.

One special item included with the VUE^2 environment (point 3) is the somewhat inappropriately named *udaemon*. This is a Tk (*wish*) script implementing an interface (GUI) to VUE^2 models.

The exact way of setting up the platform is described elsewhere (see the *Installation* document). Here we shall briefly comment on some VUE^2-specific aspects.

Suppose that the three packages have been unpacked at the same level (e.g., in the user's home directory). This is the recommended way of installing the platform, as outlined in *Installation*. They fill three separate source trees rooted at directories *VUEE*, *PICOS*, and *SIDE*.

3.1 Notes on configuring SMURPH/SIDE with VUE^2

In the present version of the platform, this step is performed automatically by the *deploy* script (see *Installation*). It can be done manually in an emergency. The procedure is basically identical to setting up SMURPH/SIDE for stand-alone operation (which is described in the *SMURPH* manual), except for declaring the VUE^2 library to be searched automatically by the SMURPH compiler for **`#included`** files.

Thus, the relevant fragment of your conversation with *maker* (the SMURPH installer) when installing the package should resemble this:

```
Now, please enter the list of paths to 'include' libraries
(which can be absolute or relative to your home directory), each
path in a separate line. Enter an empty line when done. This
path:
```


```
        /home/pawel/SIDE/Examples/IncLib

is standard and need not be specified.  If you want to exclude
the standard path, enter '-' as the only character of an input
line.
VUEE/PICOS
```

where the last line (in italics) is your response (it assumes that VUE^2 has been unpacked in the home directory). You can specify the complete path (beginning with `/`) if in doubt. Note that the default suggested by maker is retained, and one more (VUE^2-specific) include library is added. The standard (default library) is also needed as it provides the models of wireless channels used by VUE^2.

Defaults can be selected for the remaining installation parameters. The monitor, needed for the Java DSD applet is optional. The monitor connection of SMURPH is not required for interfacing VUE^2 models to external daemons, and its only practical advantage is for internal monitoring of models under intricate debugging.

You have to assign a non-default name for the *mks* program generated by *maker*, i.e., the actual compiler of VUE^2 models. That name must be one of *vuee*, *vue*, *vue2*. This is required for two reasons. First, PiComp will search for the SMURPH compiler under one of these three names. Second, when called under one of these names, the SMURPH compiler will recognize that it is running for VUE^2 and will automatically select the right set of parameters (which otherwise would have to be specified explicitly). In particular, the compiler automatically forces –L and –W, i.e., disables the (redundant) models of wired channels[3] and selects the visualization mode of SMURPH. Note that all VUE^2 models require –W to compile (while –L is optional).

It is possible to have a single installation of SMURPH/SIDE to be used for VUE^2 as well as for other purposes, i.e., simulation of network protocols. Having run *maker* once to create the *vuee* compiler, you can run it again possibly selecting another configuration of *include libraries* and assigning the standard (or different) name to the *mks* compiler. Depending on which compiler version is invoked, the corresponding *include libraries* will be referenced and the respective set of options will be applied. Note that the alternative version of the SMURPH compiler can also be created after the automatic setup by the *deploy* script.

3.2 Notes on setting up VUE^2

The VUE^2-specific components unpack into directory *VUEE*. It is convenient if this directory occurs at the same level as *PICOS*. The procedure of setting up this component is simple and consists in providing links from *VUEE/PICOS* to some files in the *PICOS* tree. Those files correspond to those elements of PicOS that have been made VUE^2-compliant, i.e., can be used directly by SMURPH as fragments of the VUE^2 model. In particular, *tcv.c* (VNETI) is one of those files.

If you want to perform this step by hand (as opposed to letting *deploy* do its job), you should move to *VUEE/PICOS* and execute *./mklinks* in that directory. This will only work if that directory occurs at the same level as *VUEE*. If this is unsuitable for whatever reason, you can edit the *mklinks* script specifying the correct path to *PICOS*.

Formally, the proper order of unpacking and installing the three systems is PICOS, VUEE, SIDE/SMURPH. This is because *VUEE* needs links to *PICOS* and *SMURPH* needs *VUEE/PICOS* as the include library.

[3] Wireless and wired channel models can coexists, if needed. The system is capable of modeling hybrid networks.

3.3 Legacy praxes

The old way of maintaining VUE^2-compliance of praxes, before PiComp, involved elaborate sets of macros, headers, and rules that must have been obeyed by the programmer to make sure that the same set of source files could be compiled for a target device (basically by the C compiler) as well as into a SMURPH model. There still exist praxes built according to those rules. To compile them, you just move into the praxis directory and execute *vuee* (or whatever name you have assigned to the SMURPH compiler). The outcome of this compilation is an executable file named *side* (*side.exe* under Cygwin). By running this file (with a suitable data file), you will execute the model. To compile the same praxis for a target board, you execute (in the same directory):

```
mkmk boardname
make
```

Note that the PicOS compiler (*mkmk*) is able to accommodate multiple praxes in the same directory (refer to the *mkmk* document for details), even though this is seldom useful when viewed solely from PicOS's angle. In the context of VUE^2, this is needed in situations when a single VUE^2 model must accommodate multiple praxes. Such praxes are usually strongly related (albeit different), and then it may make sense to keep them in the same directory also for PicOS.

It is recommended that any legacy praxis that has to undergo non-trivial enhancements, extensions, or modifications be first converted to the new format (see the *PicOS* document). The details of the old format in the context of maintaining VUE^2-compliance of praxes can be found in older versions of this document (retrievable from the GIT repository of VUEE). Refer to the *PiComp* document for a discussion of the VUE^2-compliance issues for praxes adhering to the new format.

3.4 Praxis options

File *options.sys* used by PicOS to select the optional components of the PicOS kernel and some of its parameters is also included and interpreted by VUE^2. Most of the PicOS options, however, have no meaning in VUE^2. For example, items like selecting the CPU clock are blatantly useless in a VUE^2 model. Also note that options selecting the UART rate, the LBT parameters for the transceiver, and so on, are settable in the input data set (on the per-node basis – see section 4.3), so VUE^2 does not look for them in *options.sys*.

One PicOS option that retains a part of its meaning in *options.sys* is radio module selection. It is not needed as much by VUE^2 as by the networking components of PicOS (*net.c*) that compile with VUE in their original versions. At present, you can only select CC1100 and DM2200 (because *phys_cc1100* and *phys_dm2200* are the only RF modules present in the VUE^2 library). This is very easy to amend, should a need arise.

Note that there is no way to tell VUE^2 that its should make the node fit a particular board (as described in the *BOARDS* directories). Again, this kind of specification for VUE^2 belongs in the input data set.

One important use of *options.sys* for VUE^2 is to describe the coarse-grained configuration of components from the VUE^2 library to be compiled into the node class. These components are specified via constants whose values are irrelevant (what matters is the definition or its absence). When inserted into *options.sys* such definitions can be encapsulated into:

```
#ifdef __SMURPH__
...
#endif
```


but they don't have to, as those constants are not interpreted under PicOS. Here is the complete list of options:

VUEE_LIB_PLUG_NULL	compiles in the NULL plug for VNETI
VUEE_LIB_PLUG_TARP	compiles in TARP
VUEE_LIB_XRS	compiles in the library for External Reliable Scheme over UART
VUEE_LIB_OEP	compiles in the Object Exchange Protocol library
VUEE_LIB_LCDG	compiles in the Nokia LCD display support (N6100P)

For example, this sequence in *options.sys*:

```
#define       VUEE_LIB_PLUG_NULL
#define       VUEE_LIB_PLUG_XRS
```

makes sure that support for the NULL plugin and XRS is compiled in.

Note that the compiled-in structure of the base node type (class PicOSNode) is the same for all nodes, even if the model consists of multiple praxes. This is generally not a problem because the praxis code need not use those components that it doesn't care about.

4 Input data

The input data file parameterizing a VUE[2] model follows an XML format. Below is a complete sample data file describing a three-node network:

```
<network nodes="3" radio="3" port="2234">
    <grid>0.1m</grid>
    <channel>
        <shadowing bn="-110.0dBm" syncbits="8">
              RP(d) = received power at distance d
              XP    = transmitted power
              X     = lognormal random Gaussian component
              ============================================================
              RP(d)/XP [dB] = -10 x 3.0 x log(d/1.0m) + X(1.0) - 38.0
              ============================================================
      </shadowing>
      <cutoff>-120.0dBm</cutoff>
      <ber>
              Interpolated ber table:
              ======================
                SIR           BER
              50.0dB        1.0E-6
              40.0dB        2.0E-6
              30.0dB        5.0E-6
              20.0dB        1.0E-5
              10.0dB        1.0E-4
               5.0dB        1.0E-3
               2.0dB        1.0E-1
               0.0dB        2.0E-1
              -2.0dB        5.0E-1
              -5.0dB        9.9E-1
      </ber>
      <frame>12 0</frame>
      <rates boost="yes">
              0      9600            6.0dB
              1      38400           0.0dB
              2      200000        -10.0dB
```


```
   </rates>
   <power>
          0      -30.0dBm
          1      -15.0dBm
          2      -10.0dBm
          3       -5.0dBm
          4        0.0dBm
          5        5.0dBm
          6        7.0dBm
          7       10.0dBm
   </power>
   <channels number=”255”>
          separation
          20dB 29dB 38dB 46dB 54dB 62dB 70dB 78dB 86dB 94dB 102dB
   </channels>
   <rssi>
          0      -202.0
          255      53.0
   </rssi>
</channel>
<nodes>
   <defaults>
       <memory>1124 bytes</memory>
       <processes>16</processes>
       <radio>
          <power>7</power>
          <rate>1</rate>
          <boost>1.0dB</boost>
          <channel>4</channel>
          <preamble>32 bits</preamble>
          <lbt>
                 delay                8msec
                 threshold            -109.0dBm
          </lbt>
          <backoff>
                 min                  8msec
                 max                  303msec
          </backoff>
       </radio>
       <uart rate="9600" bsize="12" mode=”direct”>
          <input source="socket"></input>
          <output target="socket" type="held"></output>
       </uart>
       <eeprom size="524288" clean="00">
          <chunk address=”0”>00 01 02 04 08</chunk>
          <chunk address=”8192” file=”eblock.bin”></chunk>
       </eeprom>
   </defaults>
   <node number="0">
       <location>1.0 4.0</location>
       <preinit tag="ESN" type="lword">0x8000ff01</preinit>
   </node>
   <node number="1" start=”off”>
       <location>1.0 10.0</location>
   </node>
   <node>
       <eeprom size=”10485760,20480” image=”node2_eeprom.img”
          clean=”00” erase=”page” overwrite=”no”
          timing="0.000004,0.000006,
                  0.000004,0.000034,
                  0.000030,0.000038,
                  0.000005,0.040000">
       </eeprom>
```


```
            <iflash size="512,2" clean="ff"></ifash>
            <sensors>
               <input source="socket"></input>
               <sensor vsize="2" delay="0.05">4095</sensor>
               <sensor vsize="2" delay="0.1,0.5" init="3">999</sensor>
               <actuator vsize="4" delay="0.01" init="0"></actuator>
            </sensors>
            <location>1.0 10.0</location>
            <ptracker>
                <output target="socket"/>
                <module id="cpu">0.3 0.0077</module>
                <module id="radio">0.0004 16.0 30.7 30.7</module>
                <module id="storage">
                       0.030 0.030 10.0 15.0 17.0 16.0
                </module>
                <module id="sensors">0.0 2.5</module>
            </ptracker>
         </node>
      </nodes>
      <roamer>
         <input source="string">
               R 2 [1.0 1.0 21.0 24.0] [0.5 1.8] [1.0 6.0] -1
         </input>
      </roamer>
      <panel>
         <input source="string">T 10.0\nO 1\nT +5.0\nF 1</input>
      </panel>
</network>
```

Illustration 2: A sample data file.

Each data file describes a single <network>. The first attribute of the <network> element (*nodes*) is mandatory and specifies the total number of nodes (of all types). The second attribute (*radio*), if present, specifies explicitly the number of nodes that will be interfaced to the RF channel. If this attribute is missing, it defaults to the total number of nodes, unless the data set contains no specification of a radio channel, in which case it defaults to zero. These numbers remain fixed during the execution, but the nodes may exhibit highly dynamic behavior including mobility.

One more (optional) attribute of <network> is *port*, which can be used to assign a non-standard port number to the socket opened by the program for connections from agents (see Section 7).

Some input elements include numbers. Typically, besides numbers, such an element may contain non-numeric text, which is ignored (treated as a comment). For example, the only relevant items from the <shadowing> element in <channel> are the numbers -10, 3.0 ,1.0, 1.0, 38.0. Also, any non-numerical characters in the otherwise numerical arguments of elements (like the letters dBm in "-110.0dBm") are ignored. Thus, the equivalent comment-free specification of <shadowing> is:

```
<shadowing bn="-110.0" syncbits="8">-10 3.0 1.0 1.0 38.0</shadowing>
```

Note that the minus sign apparently preceding 38.0 has been ignored: this is because of the space separating it from the first digit.

In the subsequent sections, we shall discuss the sub-elements of <network>.

4.1 Grid

This optional element has no arguments, and its text must include a single (floating point) number. This number specifies the granularity (in meters) of the coordinate grid for node deployment and movement. By default, the value of grid is 1.0, which means that node location will be rounded to full meters. This parameter directly determines the discrete granularity of virtual time in SMURPH (the ITU), which is equal to the amount of time required for a radio signal (propagating at 299,792,458 m/s) to cross the grid unit.

4.2 Channel

This element describes a radio channel. It used to be mandatory, but, as of release R090526A, is optional, which means that you can also model in VUE[2] nodes that are not equipped with a radio interface. If no radio channel is present in the data set, then no nodes are interfaced to the radio, which also means that a radio attribute of <network> specifying a nonzero number of nodes is illegal.

It is anticipated that the population and format of sub-elements of <channel> will expand as new types of channel models are added to the library. The element itself takes no arguments, and the exact characteristics of the channel are described by the sub-elements. The only channel model available at present is *shadowing*, whose propagation properties are described by two numerical arguments of the <shadowing> element and five numbers that must occur in its body. Argument *bn* specifies the assumed level of background noise in dBm, and *syncbits* is the minimum number of correctly received preamble bits that a receiver must perceive in front of a recognizable packet. Both arguments are required (they have no defaults).

Signal attenuation in the shadowing channel model is described by the following formula:

$$\left[\frac{P_r(d)}{P_r(d_0)}\right]_{dB} = -10\beta\log\left(\frac{d}{d_0}\right) + X(\sigma_{dB})$$

where $P_r(x)$ is the received signal level at distance x, d_0 is a (intentionally small) reference distance, β is the loss exponent and X is a Gaussian random variable with zero mean and standard deviation of σ_{dB}. For our purpose, we transform the formula to give us signal attenuation at distance d, which we can directly plug into the respective assessment method in SMURPH:

$$\left[\frac{P_r(d)}{P_x}\right]_{dB} = -10\beta\log\left(\frac{d}{d_0}\right) + X(\sigma_{dB}) - L(d_0)$$

where P_x is the transmission power and $L(d_0)$ is the calibrated loss at the reference distance d_0. This formula is is only meaningful when $d > d_0$. It is tacitly assumed that transmission at a shorter distance incurs the same attenuation as a transmission at the reference distance d_0. This is exactly the formula from the <shadowing> element on page 8, i.e.,

```
RP(d)/XP [dB] = -10 x 3.0 x log(d/1.0m) + X(1.0) - 38.0
```

with the parameter values: $\beta = 3.0$, $d_0 = 1.0\text{m}$, $\sigma_{dB} = 1.0$ and $L(1\text{m}) = 38\text{dB}$. This formula is used do determine the signal level at a receiving node N_r.

The <cutoff> element specifies a single floating point number interpreted as the so-called cut-off signal level in dBm. This is a threshold for the received signal level below which it can be safely assumed that there is no signal at all. This value is used to calculate the cut-off distance restricting the population of neighborhoods in the model, which may impact the execution speed. This element is mandatory.

All signals arriving from multiple transmitters within the cut-off range of N_r combine additively. The SIR of a single signal perceived by N_r is equal to the value of that signal level at N_r divided by the sum of all the other signals arriving at N_r augmented by the ubiquitous background noise (the *bn* argument of <shadowing>). Then, the table from the <ber> element is consulted to determine the probability of a bit error. That table is an array of numbers occurring in pairs. The numbers in the first column must be decreasing, while those in the second column must be non-decreasing (normally they are increasing). This is a discrete specification of a function that translates the signal to interference ratio (SIR) at the receiver into a bit error rate (BER), i.e., the probability that a single bit is received in error. The smooth BER function for all possible values of SIR is obtained by interpolating the table. If the SIR is greater than or equal to the first value in the table, then the bit error rate is determined by the first entry (for example, it never decreases below 10^{-6} in the channel described in Illustration 2). If it less than the last value, then the bit error rate is 1, i.e., reception is impossible.

The bit error rate applies to the so-called “physical bits,” which are also used in determining the effective transmission rate. The three numbers within the <frame> element stand for the *bit* rate, *bits* per byte, and the number of extra *bits* needed to frame a complete packet. The rate parameter determines the number of physical bits transmitted per second. The second parameter of that element translates physical bits into logical bits by indicating the number of physical bits in one byte (octet). This mapping accounts for the encoding, e.g., 6-bit symbols encoding 4-bit nibbles. The last parameter (also expressed in physical bits) covers any special (non-preamble) components of the frame that are invisible to the receiver software but contribute to the overall length of a transmitted and received packet, e.g., a start symbol.

Refer to the SMURPH manual for an explanation of the dynamics of the interference model. The assessment method responsible for detecting the beginning of a packet at a receiver (*RFC_bot*) checks if at least *syncbits* (physical) bits of the preamble (see page 8) immediately preceding the first bit of the actual packet have been received without an error, according to the bit error rate calculated as explained above. The second assessment method (*RFC_eot*), determining the final success of a packet reception, is not used in the channel model. Instead, the receiver invokes *RFC_erd* to await the first bit-error event in the packet. The (reasonable) simplification assumed in the model is that the first error bit will render its symbol invalid, which will interrupt the reception. This is warranted by the fact that all the legitimate symbols used by DM2200 (and most other RF modules) have the property that flipping a single bit renders the symbol invalid.

Clearly, the <channel> element describes a bit more than just the radio channel. Some of the parameters there are related to the characteristic of the RF modules used by the nodes. Although, arguably, it might make sense to associate those attributes with individual transceivers (and this may happen in the future), the present description favors networks based on the same (or similar) equipment used by all nodes (at least within the realm of a single “channel”). It is possible to have multiple channels possibly interfaced by nodes acting as gateways.

Owing to the fact that all RF drivers in PicOS share practically the same logical structure, it makes sense to encapsulate their hardware-specific properties into parameters that can be set in the input data file. This way, functions like *phys_cc1100, phys_dm2200* point to

essentially the same driver model, while the proper setting of the channel parameters in the input data should bring that model close to the intended hardware.

These sub-elements of <channel>: <power>, <rates>, and <channels> describe the set of options for power setting, bit rates, and channel numbers available to the nodes. In particular, the first column of numbers in the <power> element lists the discrete values usable by the praxis as arguments of *PHYSOPT_SETPOWER* with the corresponding settings of the actual transmission power for the RF module (in dBm) appearing in the second column. In a similar way, <rates> declares the selection of bit rates available to *PHYSOPT_SETRATE*. If the *boost* attribute of the <rates> element is *yes* (as in Illustration 2), then the third column of numbers specifies rate-specific factors (boosts) applied to the signal at reception. This way, different rates may result in different effective transmission ranges. If the *boost* attribute is absent (or different from *yes*), the specification consists of two columns and there are no rate specific boost factors (equivalent to 0dB boost for all rates).

If a <channels> element is present, it describes the number of channels settable by the praxis with *PHYSOPT_SETCHANNEL*. The optional sequence of FP numbers appearing within the text of that element refers to channel separation in dB. Normally, these numbers are increasing: the first number gives the separation between two adjacent channels, the second between channels n and n+2, and so on. The separation becomes infinite when the numbers run out. In particular, if no numbers are specified at all, the separation between any pair of different channels is infinite.

Elements <power> and <rates> are mandatory. In a simple case, e.g., if there is only one available power setting, the <power> element may contain a single FP number: the power setting with the implied index of zero. On the other hand, <channels> is optional. If absent, there is only one channel shared by all nodes, and its number is zero.

The optional <rssi> element describes the way of transforming the received signal strength into RSSI indications (appended at the end of a received packet). If no <rssi> element occurs within <channel>, no RSSI indications will be returned to the praxis (the corresponding packet byte will be always zero). The first column of numbers in the <rssi> element refers to the indications returned to the praxis, while the numbers in the second column describe the actual received signal levels in dBm. Values in between will be interpolated.

4.3 Nodes

This element describes the configuration of nodes in the network. Each of its sub-elements, except <default>, provides a set of parameters for one specific node. The expected contents of <defaults> are the same as for <node> and describe the default setting of parameters for all nodes. If a given parameter is not explicitly mentioned within a <node> element, its <default> setting is assumed for the respective node.

A <node> accepts three optional attributes. One of them (the keyword *number*) specifies the node number. The node numbers are internal SMURPH identifiers of the *stations* implementing the nodes. Although nodes can be specified in any order in the data file, the resultant numbering of nodes must be continuous and start from zero. The total number of node definitions (<node> elements) in a data set must be equal to the *nodes* attribute of <network> (page 8).

An explicit number attribute of a <node> element, if present, makes it obvious how to match the definition to an actual node number in the network model. If the number attribute is absent, it is assumed that the definition has an implicit number tag equal to the number of the previous definition + 1. If the first <node> definition has no number attribute, it is

assumed to refer to node 0. Note that it is legal for the input data set to contain non-contiguous chunks of node definitions. In any case, to be correct, they should exhaust all node numbers (between 0 and *nodes*-1) and do not attempt to describe the same node more than once.

Another attribute of <node> (keyword *type*) is a piece of string identyfying the node type. The value of this attribute is useful in models with multiple praxes.

One more (optional) attribute of <node>, which is also applicable to <defaults>, is *start*, which can be “on” or “off”, with “on” being the default. If the value of start is “off” for a node, then the node will not be started automatically. Such a node will have to be started explicitly, either from a panel process (described by a <panel> element in the input data – see Section 6) or by an external agent.

Here is the list of sub-elements of <node> (and also <defaults>). The nontrivial elements are discussed later in separate sections.

<memory> ... *memory size in bytes* ... </memory>

This element declares the amount of memory (standard RAM) available at the node for *malloc*. This is equal to the physical size of RAM at the microcontroller minus the combined size of global variables (with provision for alignment). PicOS reports this amount upon startup as the so-called leftover RAM.

<processes> ... *process number limit* ... </processes>

This element declares the size of the process table, i.e., sets the limit on the maximum number of PicOS processes that may be present at the same time. If the element is absent, there is no explicit limit on the number of processes.

If a node is equipped with radio interface, then its declaration should include a <radio> element, which, in turn, should mention the following attributes:

<power> ... *power index* ... </power>

This element declares the initial setting of the transmission power with reference to the <power> table specified for the channel. The value must be one of the indexes occurring in the first column of the channel's <power> table. If the element is missing, the lowest index from the <power> table is implicitly assumed.

<rate> ... rate *index* ... </rate>

This element declares the initial setting of the transmission rate according to the <rate> table for the channel. The value must be one of the indexes occurring in the first column of the channel's <rate> table. If the element is missing, the lowest index from the <rate> table is implicitly assumed.

<channel> ... *channel number* ... </channel>

This element declares the initial channel setting for the RF module. The value must be a valid channel number according to the <channels> specification for the channel. If the element is missing, channel 0 is assumed by default.

<boost> ... *receiver gain* ... </boost>

This element declares the receiver gain (boost) in dB, which is 0.0dB by default.

<preamble> ... *preamble length in physical bits* ... </preamble>

This element specifies the packet preamble length in physical bits to be inserted by the node's transmitter in front of every packet.

```
<lbt> ... delay ... threshold ... </lbt>
<backoff> ... minimum ... maximum ... </backoff>
```

These parameters (four numbers) are used by the collision avoidance procedure of the node's transmitter. For <lbt> (Listen Before Transmit), the first (integer) number gives the time interval (in milliseconds) during which the transmitter will pause before a spontaneous transmission while monitoring the signal level. If the level is above the threshold (in dBm), the transmitter will back off randomly and try again. The back-off delay is between the minimum and maximum specified in the <backoff> element (both numbers are in milliseconds and must be integers).

The total number of nodes whose specification includes a <radio> element must match the number of nodes declared with the <network> element as being equipped with a radio interface. Note that this is usually the total number of nodes in the network, unless the <network> element has a non-trivial *radio* attribute. if the <default> element specifies a radio interface (has a <radio> sub-element), then any node devoid of <radio> is automatically assigned the default specification. If you want to explicitly indicate that a particular node is NOT equipped with a radio, use an empty <radio> element, e.g.,

```
<radio></radio>
```

This element is mandatory for a node equipped with radio:

```
<location> ... x ... y ... </location>
```

and it must be explicitly present in every such a <node>. There can be no default for a node's location: the <location> element is ignored if occurring in <default>. The element assigns an initial location to the node as a pair of coordinates in meters (floating point numbers). The coordinates can be arbitrary as long as they are non-negative. Nodes can only be deployed in the right upper quarter of the infinite Cartesian plane.

A radio-less node need not have its location specified (although it isn't forbidden). The default location of such a node is (0, 0).

The following <node> sub-elements are optional: <uart>, <pins>, <leds>, <eeprom>, <iflash>, <preinit>. They first five augment the nodes with optional components (like UARTs), which can be legitimately absent. The last one provides a way of initializing selected node attributes. All these elements are discussed below, each in a separate section.

4.4 The UART

Each node may be optionally equipped with a UART, whose functionality, as perceived by the praxis, accurately mimics two most popular ways of accessing the UART in PicOS: 1) direct interface (operations *ser_in*, *ser_inf*, *ser_out*, *ser_outf*, *ser_outb*), and 2) packet-style interface with the UART being viewed as a PHY for VNETI (TCV) (modes 'N' and 'L' described in *Serial.pdf*). This option is selected with the *mode* attribute of the <uart> element, which can be one of the following:

“direct”	(or “d”)	selecting the default direct interface to the UART
“npacket”	(or “n”)	selecting the 'N' mode interface over TCV
“ppacket”	(or “p”)	selecting the 'P' mode interface over TCV
“line”	(or “l”)	selecting the 'L' mode (line) interface over TCV

With the direct mode, the praxis can access the UART via the collection of “ser_” operations listed above. With the packet mode, the praxis can open the UART PHY by executing *phys_uart* (as described in *Serial.pdf*). Then it can associate a plugin with the UART PHY and use the interface essentially in the same way as an RF module. Finally, the line ('L') mode makes it possible to use the VNETI packet interface with a straightforward ASCII-oriented appearance of the UART at the other end.

The UART description in the input data assigns a bit rate to the UART, declares the sizes of two buffers to be used by the modeled driver, and determines what happens to the output and where the input will be coming from. The first three values are provided as attributes of <uart>, e.g.,

```
<uart rate="9600" bsize="12,8">
```

with the *rate* attribute being mandatory and *bsize* being optional. The first of the two *bsize* numbers gives the length of the input buffer,[4] and the second one declares the size of the output buffer. The default buffer size is 0 (in both cases), which effectively corresponds to “no buffer.”[5] This is assumed for both buffers, if no *bsize* attribute is present in <uart>, or for the output buffer, if only one number is provided, e.g.,

```
<uart rate="9600" bsize="12">
```

The UART's interface to the real world is described by the <input> and <output> elements. The options are:

Local file or device:

```
<input source=”device”>device or file name</input>
<output target=”device”>device or file name</output>
```

The body of each element is stripped of any initial and trailing white spaces and the remainder is interpreted as a file name path relative to the directory where the model (the *side* program) has been invoked.

If the names in fact refer to files, than they should be different to make sense. On the other hand, they may represent the same device, e.g., a TTY terminal window. In that case, the UART may directly interact with the user.

The (input) characters arriving from a file/device, as well as those written by the praxis to the UART and sent to a file/device, are not preprocessed in any way (using the UNIX terminology, we would say that the interface is “raw”). The assumed interpretation of lines in PicOS, for the ASCII-oriented UART access functions *ser_out, ser_outf, ser_in, ser_inf,* is that an output line ends with CR+LF, and an input line must end with at least one of those characters, with any sequence of CR and/or LF characters at the end interpreted as a single end of line.

Both <input> and output elements accept an optional *coding* attribute, which can be “hex” or “ascii”, with “ascii” being the default, e.g.,

```
<input source=”device” coding=”hex”>
```

[4] This corresponds to a compilation parameter for PicOS.

[5] In fact a single-byte buffer is used by the model in that case, which corresponds to the hardware UART register on the microcontroller. This has the same meaning as the buffer size of zero in PicOS. Also, the MSP430 UART driver in PicOS uses 0 for the output buffer size (there is no parameter to change that.

With the “hex” coding, the input is assumed to be a sequence of bytes expressed as pairs of hexadecimal digits. Whenever a next byte is read from the UART, the emulator will look up in the file the next character looking like a hexadecimal digit (skipping all other characters). Then, if the following character is also a hexadecimal digit, the two will be decoded into a byte and returned as the read character. Otherwise, the program will be aborted with an error message. For output, the “hex” coding produces a sequence of 2-digit hexadecimal representations of the output bytes separated by spaces.

The <input> element accepts an optional *type* attribute, whose value can be “timed” or “untimed”, with the latter being the default, e.g.,

```
<input source="device" type="timed">
```

For the “timed” type of input, the input file must be organized into records looking like this:

```
time { input data }
```

where time is a floating point number describing the time when the record will become available for input. If preceded by a '+' sign, the number describes the interval in seconds from the availability time of the previous record (or from 0 if the record starts the input sequence). If there is no '+', the number represents the absolute time in seconds from the beginning of run (time 0). If the time has already passed when the record is looked at by the system (i.e., the processing of previous records took more time than the record's availability time), the record becomes available immediately.

Once a record becomes available, its characters will arrive at the UART at the nominal rate specified in the <uart> element. If the praxis does not retrieve them on time, they will be lost. This is different from the “untimed” (default) operation whereby the bytes to be read by the praxis patiently await acceptance. In that case, the UART rate only determines how often they can be extracted (the minimum space between characters), but they never arrive faster than the praxis can cope with them.

Here is an example of a timed input sequence:

```
5.0 {s 4096\r\n} +4.0 {r\r\n} 3600 {s 0\r\n}
```

The string within braces may not include a closing brace unless escaped with a backslash. Generally, any character can be escaped with a backslash (including the backslash itself, which must be escaped). The escapes *r*, *n* and *t* are treated as special characters (the last one standing for a TAB). An explicit newline also stands for itself, i.e., is equivalent to *n*.

The timed mode can be combined with “hex” coding, e.g.,

```
5.0 {73 20 34 30 39 36 0d 0a} +4.0 {72 0d 0a} 3600 {73 20 30 0d 0a}
```

is equivalent to the previous sequence under “hex” coding.

Short input sequence specified directly in the data file:

The source attribute of <input> can be “string”, e.g.,

```
<input source="string">a direct sequence of bytes</input>
```

This specification is most useful in those circumstances when the node expects some short input from the UART at the beginning, e.g., to initialize the praxis. Generally, if the input part of the UART file is reasonably short, it can be inserted directly into the body of the <input> element, e.g.,

```
<input source="string">s 4096\r\n</input>
```

Note that the output may still be assigned independently to a file/device. Also, the "string" source can be combined with "hex" coding and/or "timed" mode, e.g.,

```
<input source="string" type="hex" mode="timed">
5.0 {73 20 34 30 39 36 0d 0a}
+4.0 {72 0d 0a}
3600 {73 20 30 0d 0a}
</input>
```

Defining a <default> UART whose input is mapped to a file is usually not a good idea, even if it works in principle. This is because, for a large network, the multiple instances of the file being opened for different nodes may deplete the population of allowable file descriptors and crash the model. On the other hand, having a default UART with an immediate input string (say for a default initialization of all the nodes) makes perfect sense. The output part of such an UART can be legitimately left unspecified.

Needless to say, the above ways of providing external input to a UART, or absorbing its output, are not very convenient (in fact they are practically useless) in the packet mode. Typically, in such a case, the praxis wants to talk to some OSS program implementing some non-trivial protocol. The proper way of handling this kind of interaction is to direct the modeled UART to an external agent (like udaemon – see below) providing a link between the virtual world of the VUE^2 model and the real-life OSS program.

Remote association through an agent:

The most flexible option is to map the UART to a socket and make it possible for external agents to claim its input and output. This is accomplished by using "socket" for the *source* and *target*, e.g.,

```
<input source="socket"></input>
<output target="socket"></output>
```

Once you map one end (say input) to a socket, the other end (output) also becomes mapped to a socket (there is no way to map it differently). Thus, the second line in the above sequence is redundant (although harmless). It may may be needed, if some other attribute of the element must be included, e.g., *coding="hex"*. A specification like this:

```
<input source="device">/dev/tty</input>
<output target="socket"></output>
```

will result in an error.

The body of an <input> or <output> element specifying "socket" is always ignored. External agents connect to such a UART following a special protocol that identifies the node (see Section 7.2). Thus, there is no need to provide any more details at this stage. A socket input can also be "hex" and/or "timed". A socket output can be "hex". Additionally, an <output> element specifying *target="socket"* may also include *type="held"* among its attributes, e.g.,

```
<output target="socket" type="held"></output>
```

The meaning of this setting is that the initial output written to the socket is saved and will be presented to the agent upon its (first) connection. This way you can make sure that whatever the praxis writes to the UART is never lost. This is important because before an agent can connect to the socket, the model must be started; thus, if the node writes something to the UART immediately after startup, that output might be lost.

Partially mapped UARTs

A socket UART is flexible in the sense that an agent may connect to it and disconnect many times. If anything is written to the UART while no agent is connected to it (excepting the period preceding the first connection to a "held" socket), that output is discarded and lost. Then if the node tries to read something from a disconnected socket, it will simply receive no data. This is the same as if the real UART were disconnected from the device.

For a device-mapped UART (and also for a UART whose input end is mapped to a string), it is legal to leave one end unmapped. For example, when writing to a UART whose output end is unmapped (but the UART is present), the output will be absorbed by the UART and discarded. An attempt to read from a UART with no input end will never return any data, but is formally legal. On the other hand, if no UART is defined at all, any attempt to reference it for I/O will trigger an error and abort the model.

You can indicate explicitly that the particular end is unmapped by saying something like this:

```
<output target="none"></output>
```

This is equivalent to simply skipping the specification. Note that a socket UART always defines two ends, even if only one end is explicitly mentioned in the declaration.

If a <node> element has no <uart> sub-element, but there is such a sub-element in <default>, then the <default> specification of UART will be assumed by the node. If the node wants to say explicitly that it has no UART, default or otherwise, the following declaration should be used:

```
<uart></uart>
```

This is the only variant of the <uart> element that requires no rate specification.

4.5 The LEDS module

For a node equipped with LEDs, you can make their status traceable or presentable to external agents. Here is a sample declaration of the LEDS module within <node>:

```
<leds number="4">
      <output target="device">led_status.txt</output>
</leds>
```

It says that the updates to the LEDs will be written to file *led_status.txt*. Each update takes one ASCII line beginning with the letter U (capital) and terminated with the newline character. The complete line looks like this:

```
U T F LLLLL ... LLL
```

where *T* is the time of the status change in seconds (a floating point number), *F* is 1 or 0, depending on whether the fast blink option is active or not (see PicOS operation *fastblink*), and each of the subsequent characters stands for the status of one LED from zero up (i.e., the length of the *L...L* string is equal to the number of LEDs). The values are: 0 – the LED is off, 1 – the LED is on, 2 – the LED is blinking. Here is an example:

```
U 10.325 0 0101
```

To make the LEDs status perceptible by external agents, use "socket" as the output target, e.g.,

```
<leds number="2"><output target="socket"></output></leds>
```

Having connected to the LEDS module via the socket interface, the agent will be receiving updates in the same format as when they are written to a file.

Note that a LEDS module description can be placed in <defaults>. It may not make a lot of sense to direct LEDs status updates of all nodes to the same file, but it is perfectly legal to declare that all nodes can have their LEDs inspected by an agent (*target="socket"*). If no agent is connected to the module, changes in the LEDs status trigger no external actions. Upon a connection, the agent will receive the current status of the LEDs, and then it will be receiving updates for as long as it is connected.

Similar to UART, to explicitly say that a node has no LEDs (overriding the default), you can use this declaration:

```
<leds></leds>
```

Specifying zero as the number of LEDs has the same effect. In that case, the <output> specification is ignored if present.

4.6 The PINS module

This module provides an external interface to the node's I/O pins, including ADC and DAC. The node-inflicted changes to the (output) pins can be sent to an agent or stored in a file. Similarly, external changes to the pin voltage can be submitted by an agent or read from a file, possibly along with their timing. Here is a sample declaration:

```
<pins total="10" adc="8" counter="1,8,8" notifier="2,8,256" dac="6,7">
      <output target="device">pins_output0.txt</output>
      <input source="device">pins_input0.txt</input>
      <status>1111111111</status>
      <values>0000000000</values>
      <voltage>0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0</voltage>
</pins>
```

which is complex enough to illustrate all sub-elements that can contribute to <pins>. Except for the trivial declaration <pins></pins>, which explicitly states that the node has no PINS module, the *total* attribute of <pins> is required, and it specifies the total number of I/O pins in the module. If the *adc* attribute is present, it declares the number of pins capable of providing input to the analog-to-digital converter. That number cannot be larger than *total*. The ADC-capable pins constitute an initial subset of all pins.

The conventions assumed for identification and access to the pins are strongly related to the standard pin interface offered by PicOS (functions *pin_read, pin_write, pin_read_adc, pin_write_dac,* etc., including pulse monitor and notifier). Some familiarity with that interface will help you understand the ideas behind the PINS model.

If the PINS module is to be equipped with the pulse monitor, which consists of two pins: the counter and the notifier, the numbers of those pins can be specified with the optional *counter* and *notifier* attributes. The full specification of each of those pins takes up to three numbers. The first number identifies the pin, and must be less than the total number of pins, the remaining two values describe the "on" and "off" debouncing intervals expressed in milliseconds. Both, counter and notifier pins can be any pins, including those capable of ADC/DAC operation. Similar to PicOS, if the counter/notifier is not running (the function is switched off by the praxis), the respective pin can be used for its standard function.

The interpretation of the debouncing parameters is as follows. If the pin gets into the triggering state (depending on the edge setting, e.g., high for edge = 1), it must remain in this

state for at least the “on” interval for the trigger to be considered valid. If the pin state changes within the “on” interval, the trigger is ignored. Similarly, once the trigger is successful, the pin must enter the opposite state for at least the “off” interval before it can be monitored for a next trigger. If a debouncing parameter is not specified (or is explicitly zero), the corresponding phase of the action is not debounced. Note that it is legal to specify a single debouncing parameter (which will be interpreted as the “on” interval, with the “off” interval assumed zero). If no debouncing parameters are specified, they are both assumed to be zero.

If any pins should provide DAC output, their numbers can be specified with *dac*. Note that only some models of MSP430 offer this functionality, which is confined to two pins.

The role of <status> is to indicate whether any of the pins in the range 0 ... *total*-1 are absent, i.e., the range includes holes.[6] The status of each pin is described by a single binary digit in the string, with 1 standing for “present” and 0 for “absent,” with the leftmost digit corresponding to pin 0. If no <status> element appears within <pins>, it is equivalent to “all ones,” i.e., no absent pins. If the string is shorter than the number of pins, the unaccounted for pins are all present by default.

The <values> element assigns initial digital values to all pins, which can be 0 or 1. Note that this assignment is only meaningful if the praxis sets the pin to input. At this stage, the declaration is equivalent to pulling the physical pin down or up via a resistor. Note, however, that it can be changed dynamically through an agent or from an input file. If the specification is absent, the pin values default to all zeros. If the string is shorter than the number of pins, the unaccounted for pins are all pulled down (initialized to 0).

Similarly, <voltages> assigns predefined analog values to the ADC-capable pins – in the natural order. When such a pin is selected by the praxis for its ADC function, this is the constant voltage that will show up on the pin for conversion. Again, that voltage should be viewed as initial as it can be changed by an agent of from an input file. If the specification is absent, the voltage defaults to all zeros. If the number of items in the list is less than the number of ADC-capable pins, the unaccounted for pins are set to voltage 0.

The <input> and <output> specifications are similar to those for UART, albeit simpler because the elements take no attributes besides “source” and “target.” The “string” source option is also available. There is a way to set up timed scripts for configurations of pin values via explicit sequences in the input data.

An input command addressed to a pins module is an ASCII line starting with one of the letters *T, P, D*. A *T* command requests a delay before reading the next command. The letter must be followed by a non-negative floating point number optionally preceded by a sign, e.g.,

```
T 12.5
T +0.01
```

In the first case, the number indicates the absolute time in seconds at which the next command should be read and interpreted. If the model is already past that time, the next command is read immediately. In the second case, the specified delay in seconds is calculated from the current moment.

A *P* command sets the digital value of a single pin. The letter is followed by the pin number in decimal followed in turn by 0 or 1, e.g.,

[6] Even though the pin numbering is purely logical (defined on the per-board basis), the numbering convention assumed in PicOS makes it possible to exclude some pins from the continuous range.

```
P 12 0
P 0 1
P 14 1
```

A *D* command assigns a voltage to the specified pin. The letter is followed by the pin number and the discretized voltage as a number from 0 to 32767 (0x7FFF) corresponding linearly to 0 ... 3.3V.[7]

An integer number can be specified in decimal (in the natural way) or in hexadecimal preceded with 0x. Initial spaces preceding the command letter are ignored. Thus, for example, the following sequence of commands makes sense:

```
T 0.5
       P 0xc 1
       P 0 0x0
       D 0xf 0x00ff
T 1.0
       D 0xf 0x0000
```

An immediate input string (the *source="string"* option) should look like the contents of a file with a series of input commands, with the command lines separated with explicit new lines or *n* sequences, e.g.,

```
<input source="string">P 1 1\nP 2 1\nT 0.01\nP 2 0</input>
```

is equivalent to:

```
<input source="string">
       P 1 1
       P 2 1
       T 0.01
       P 2 0
</input>
```

The first new line (the one preceding the first command) in the second version is stripped off by the data parser. This is not very important as empty lines in the command sequence are always ignored.

If the <output> part of the interface is configured and active, every change in the status of a pin results in a message being written to the file or sent to the agent. At the very beginning of the output sequence, there will be one special message looking like this:

```
N tt an
```

and specifying the number of pins: *tt* is the total number of pins and *an* is the number of ADC-capable pins (as specified in the attributes of the <pins> element). This line is also sent as a first message to any agent connecting to the PINS module over a socket – immediately after the connection has been established.

An actual update message starts with the letter U (capital) and consists of the following four components:

1. The time in seconds (with millisecond granularity) followed by a colon.
2. The number of the affected pin.
3. The status of this pin (a single decimal digit).
4. The pin value.

[7] No negative voltage is implemented at present.

Here we have a few examples:

```
P 16 8
U 24.556: 4 1 0
U 3600.120: 7 3 176
U 2365.667: 2 0 0
```

The pin status value is interpreted as follows:

0 – digital input pin
1 – digital output pin
2 – an ADC pin
3 – DAC pin number 0
4 – DAC pin number 1
5 – pulse monitor counter pin
6 – pulse monitor notifier pin

The output value for cases 5 an 6 is always zero. For cases 2, 3, 4, the value is a number between 0 and 32767 representing the voltage (it is the output voltage for status 3 and 4, and the input voltage for status 2). For cases 0 and 1, the value can only be 0 or 1.

When the output is directed to a file, it starts with an *N* message followed by the complete list of initial pin values. From then on, the full status of all pins can be determined by tracking the updates reflecting changes in the individual pins. In the socket case, whenever an agent connects to the module, it immediately receives an *N* message followed by the initial series of "updates" for all pins reflecting their current status and values.

4.7 Buttons

PicOS offers a standard interface to buttons, which are selected pins capable of triggering "button pressed" events. In VUE[2], buttons can be declared in the input data set, within the context of <pins>, such that pertinent changes in their voltage will translate into button pressed events perceptible by the praxis (the *buttons_action* function). Consider this sample declaration:

```
<pins number="7">
      <input source="socket"></input>
      <buttons polarity="low" timing="200">0 2 5 6</buttons>
      <values>1111111</values>
</pins>
```

With this declaration, pins number 0, 2, 5, and 6 of the total 7 pins will be used as buttons (numbered 0 through 3). Their pressed status is low, and the single debounce parameter (200) stands for the minimum number of PicOS milliseconds separating two button press events. The list of debounce values can include two more numbers: the initial delay (in milliseconds) before an automatic repeat (if the button remains pressed) and the repeat interval (also in milliseconds). If the first of these numbers is zero (or not specified), auto repeat is disabled. If the second number is zero (or absent), it is assumed to be the same as the first number.

The legitimate values of *polarity* are *low*, *high*, *0*, *1* (alternatives for *low* and *high*), with high being the default. The numbering of buttons (as seen by the praxis) is always consecutive from zero up. The list of values in the body of <buttons> is interpreted as an array mapping indexes from zero up into pin numbers.

4.8 The SENSORS module

This module provides an external interface to the node's set of sensors and actuators, which are accessible by the praxis via the PicOS functions *read_sensor* and *write_actuator*. Here is a sample definition:

```
<sensors>
        <output target="device">actuator_output0.txt</output>
        <input source="device">sensor_input0.txt</input>
        <sensor number="0" vsize="2">398</sensor>
        <sensor vsize="1" delay="0.001"></sensor>
        <sensor vsize="3" delay="0.1,0.3" init="4">15</sensor>
        <sensor number="4">257</sensor>
        <actuator vsize="2" delay="1.0,2.0">4095</actuator>
</sensors>
```

The <sensors> element has no attributes. The total number of sensors and/or actuators is determined from the element's contents and need not be announced. The arguments and definition layouts are the same for sensors and actuators.

All attributes of <sensor> and <actuator> are optional. The *number* attribute assigns a number to the sensor/actuator, which will identify it for the praxis (it corresponds to the second argument of *read_sensor/write_actuator*). If the number attribute is absent, the sensor/actuator will be assigned the next unused number. This is similar to the definition of nodes (Section 4.3). If the first sensor/actuator element has no number attribute, it is assumed to be zero. Note that sensors and actuators are numbered independently from zero. The maximum number for a sensor/actuator is 255.

The numbers of sensors/actuators defined in the data file need not cover a contiguous range, but each sensor/actuator can be defined at most once. Nonexistent sensor/actuators will trigger errors when referenced. Note that sensor number 3 in the above example does not exist, but sensor 4 is there. Holes like this may make little sense, but they are not illegal.

Attribute *vsize* determines the size of the sensor/actuator value in bytes. Legal sizes are 1, 2, 3, and 4. The default size is 4, but, if the definition specifies a range (see below), and *vsize* is absent, the size will be determined from the range: as the smallest number of bytes needed to accommodate the maximum possible value stored in the sensor/actuator. In all cases, the value to be extracted from a sensor or stored in an actuator is interpreted as an unsigned integer, which is never negative.

Size 3 is discouraged and may never be used for a real sensor/actuator. While sizes 2 and 4 are interpreted as *word* and *lword* types in an endian-independent way, it is impossible to do the same for a 3-byte number. For now, such a number is interpreted as little-endian.

Attribute *init*, if present, provides an initial value for the sensor/actuator. If that value exceeds the declared range (see below), the maximum legal value is used. If the attribute is absent, the sensor/actuator is initialized to zero.

The *delay* attribute should consist of one or two (comma separated) floating point numbers. It describes the delay in seconds incurred in reading a value from the sensor (via *read_sensor*) or storing a value into the actuator (via *write_actuator*). If there are two numbers, then the delay will be generated as a random value between them. Note that the second number must not be less than the first one. If the *delay* attribute is absent, operations on the sensor/actuator incur no delays.

If the body of a <sensor> or <actuator> element contains a strictly positive integer number, that number is interpreted as the range, i.e., the maximum value that the sensor/actuator is able to assume. The minimum is always zero. If no range is specified, it is inferred from

vsize as the maximum unsigned value that can be stored on the specified number of bytes. If neither *vsize* nor the range is present, then, by default, the size is assumed to be 4 and the range is 2^{32}-1.

The <input> and <output> specifications are exactly the same as for PINS. If the input is from a socket, then the output is implicitly assumed to be present and directed to the same socket. As for all other types of objects, It is illegal to associate one direction with a socket and the other with something else.

An input command addressed to the module is an ASCII line starting with one of the letters *T* or *S.* Similar to PINS, a *T* command requests a delay before reading the next command. An *S* command has this syntax:

```
S sn v
```

where *sn* is the sensor number and *v* its new value. The rules for encoding numbers are as for PINS. Here is a sample sequence of commands:

```
T 0.5
S 0 223
S 1 0xffe
T +1.0
S 0x2 19999
```

If the <output> part of the interface is configured and active, every change in the value of a an actuator results in a message being written to the file or sent to the agent over the socket. If the connection is over a socket (as opposed to a device), then, additionally, every change in the value of a sensor results in a similar message. The latter can be used by the remote agent as an acknowledgment of its last *S* request.

The first message sent Immediately after initiating the connection, or after node reset, looks like this:

```
N an sn
```

and tells the number of actuators (*an*) and sensors (*sn*) defined for the node. Strictly speaking, accounting for the possible holes, *an*/*sn* is the maximum number of a defined actuator/sensor + 1. This is followed by the list of initial “update” messages, each looking like this:

```
A an vvvvvvvv mmmmmmmm
```

or this:

```
S sn vvvvvvvv mmmmmmmm
```

The second message type is only present for a socket interface. The messages tell the current (initial) values of all defined actuators/sensors together with their ranges. The first number following the letter (*A* or *S*) is the decimal sensor/actuator number (between 0 and 255 inclusively), the next (hexadecimal) number is the value, and the last (also hexadecimal) number is the maximum. All actuators appear before sensors, and both types of objects are listed in the increasing order of their numbers; thus, a hole directly corresponds to a sensor/actuator that is absent.

Following the initial “update,” subsequent updates are sent without the range component, i.e., the letter (*A* or *S*) is only followed by two numbers. The second number is always in hexadecimal and consists of exactly eight (hexadecimal) digits (no *0x* prefix).

4.9 Storage modules: EEPROM, SD, information flash

VUEE allows you to model two kinds of external storage: generic EEPROM (which may also be used to mimic an SD card) and the so-called *information flash* (e.g., corresponding to the Flash Information Memory available on MSP430 processors). Both types (i.e., EEPROM and information flash) can be present simultaneously. The list of operations applicable to them can be found in the *Storage.pdf* document (coming with PicOS). Operations on SD cards are mapped to the corresponding operations on EEPROM. Note that you cannot have them at the same time (as at most one EEPROM module can be defined per node). There is no access to general code flash in VUEE (operations *cf_write*, *cf_erase*). These operations are completely void in VUEE: *ee_open*, *ee_close*, *ee_panic*, *sd_open*, *sd_close*, *sd_panic*, *sd_idle*. Those of them that return values behave as if they always succeeded.

EEPROM is declared with the <eeprom> element (see Illustration 2). The attributes of that element specify the parameters of the module, while the body may contain initializers (<chunk> elements) that pre-load values into specific fragments of the storage. Here is the list of attributes of <eeprom>:

`size=`"*n*" or **`size=`**"*n,m*" (where *n* and *m* are integer numbers)

This is the only one attribute that is mandatory, in the sense that its absence means that EEPROM is undefined. A possible application of such a declaration, e.g.,

```
<eeprom></eeprom>
```

is to indicate that the node should have no EEPROM even if there is one in <defaults>. Note that no partial inheritance from <defaults> takes place, i.e., if an EEPROM module is defined at a node at all, then all its parameters must be specified with that definition (the absent ones assume global default values that need not match those in <defaults>).
The first number of *size*, *n*, specifies the total size of the storage in bytes. The second (optional) number, *m*, gives the number of pages (or blocks). This is important if you want to make sure that the erase operation applies to an entire number of pages/blocks, in which case *m* provides the requisite grain. If the number of pages, *m*, is present, then the total size *n* must divide evenly by *m* and the result (the page size) must be a power of two.

`clean=`"*xx*"

The value is a single-byte hexadecimal number which specifies the contents of an erased byte. If the attribute is missing, the value defaults to **`FF`**.

`erase="byte"` or **`erase="page"`**

The attribute specifies the granularity of erase, i.e., whether you can erase individual bytes (this is the default) or pages. The latter case only makes a difference when the page size is nontrivial.

`overwrite="yes"` or **`overwrite="no"`**

The attribute indicates whether overwriting an already written byte (without a prior erase) produces the correct result (the first case, which also happens to be the default). In the second case, it is assumed that overwriting zeros with ones has no effect.

`timing=`"*rl,ru,wl,wu,el,eu,sl,su*"

This attribute specifies the timing parameters that determine the delays incurred by operations *ee_read*, *ee_write*, *ee_erase*, *ee_sync.* For those operations that admit a state

argument (and can possibly block), i.e., all of the above except for *ee_read*, those parameters determine the amount of time elapsing until the respective operation unblocks after its start. This will translate into an actual delay modeled by VUE[2]. For *ee_read* (which has no state argument and never blocks, VUE[2] cannot simulate the delay. This also happens when the state argument of *ee_write*, *ee_erase*, or *ee_sync* is **WNONE**. In that case, the delay is only applied to calculating the power usage by the power tracker (see below) and otherwise ignored. In particular, SD card operations (which do not accept the state argument) behave like the corresponding EEPROM operations with the state argument being **WNONE**.

The timing parameters are floating point numbers providing the lower and upper limits (in seconds) for the four operations (in the listed order) per one byte, except for ee_sync (which takes no length argument). The delay in each particular case is determined as a uniformly-distributed random number between the bounds. If not all numbers have been specified, the missing ones (from the end) are assumed to be zeros, which has the effect of not modeling any delays for the corresponding operation(s).

image="*filename*"

This attribute says that there is a file backup for the storage. If *filename* refers to an existing file, this file will be opened and its contents will appear as the initial contents of the storage. If the file is larger than the formal size of the storage, it will be truncated down to size. If it is shorter (in particular, if its length is zero or it doesn't exists), it will be extended and filled up with “erased” bytes (depending on the *clean* attribute). Whenever some bytes are written to the EEPROM, those bytes will be stored in the file, which can be subsequently viewed, archived, and so on after the experiment.

It is not illegal to use the same *filename* for multiple nodes and may make perfect sense if the storage is used read-only. This is not checked, however, so you will mess things up if multiple nodes write to the same backup file. As a precaution, it is formally illegal to use this attribute if the <eeprom> element occurs in <defaults>.

EEPROM can be preinitialized with a sequence of <chunk> elements appearing within the body of the <eeprom> element (see Illustration 2). One mandatory attribute of such an element is *address* providing the offset into the EEPROM where the specified chunk of bytes goes. The bytes themselves can be listed as the body of <chunk>. For example, this chunk:

```
<chunk address="8192">
      0 1 2 3 FE 0xAC 0 0 0 11
</chunk>
```

will preset 10 bytes starting at address 8192 with the specified contents. The numbers appearing within the body of a <chunk> are all in hexadecimal (the **0x** prefix is optional) and each of them stands for exactly one byte.

Another possibility is to read the contents of a file into an EEPROM fragment, e.g.,

```
<chunk address="0" file="fonts.nok"></chunk>
```

In this case, the body of the <chunk> element is ignored, and the entire contents of the specified file are written into the storage at the specified address.

A single <eeprom> definition may include multiple chunks of different kinds. They can overlap (being applied in the order of their occurrence), but none of them is allowed to extend beyond the formal limits of the storage. An attempt to do that will be signaled as an error.

Note that initialization with chunks is compatible with backing files. For example, this definition is legal:

```
<eeprom size="536870912,1048576"
    image="myimage.bin" clean="00" overwrite="no">
      <chunk address="0">ba ca de ad"</chunk>
      <chunk address="16384" file="picture.nok">
</eeprom>
```

and its outcome is intuitively clear. The contents of the initializing chunks (wherever they come from) will be written into the backing file.

Note that EEPROM (and information flash), unlike other node components, are not re-initialized after the (modeled) reset.

Functionally, the information flash is a subset of EEPROM. The <iflash> element used for its definition accepts the same attributes as <eeprom> (with the same meaning), except for *erase*, *overwrite*, and *timing*. Even though *clean* is admissible, *overwrite="no"* is forced for <iflash>, as is *erase="page"*. Similar to EEPROM, information flash can be initialized (the body of <iflash> may include <chunk> elements) and/or backed to a file.

4.10 Nokia LCD display N6100P

An <lcdg> element appearing as part of <node> (or <defaults>) equips the node with the model of a graphic LCD. The *type* attribute of this element determines the display model. At present, the only legitimate type is *n6100p*; thus,

```
<lcdg type="n6100p"></lcdg>
```

incorporates a virtual N6100P into the node. The body of the <lcdg> element should be empty. A sequence without the type attribute, i.e.,

```
<lcdg></lcdg>
```

indicates that the LCD model is absent and can be used to nullify locally a global declaration appearing in <defaults>.

4.11 Power tracker

The power tracker module allows you to maintain a record of energy usage by a node and, in particular, estimate the average amount of current drawn by the node within a certain time period. At present, the built in functionality of the power tracker covers: the CPU states (low power versus normal), the RF module, non-volatile storage (EEPROM, SD card), and sensors.

Power tracker parameters are specified on a per-node basis (see page 10) and may appear as part of a <node> element or in <defaults> (with the standard meaning). Here is a sample specification involving all four components that are traceable at present:

```
<ptracker>
    <output target="socket"/>
    <module id="cpu">0.3 0.0077</module>
    <module id="radio">0.0004 16.0 30.7 30.7</module>
    <module id="storage">
         0.030 0.030 10.0 15.0 17.0 16.0
    </module>
    <module id="sensors">0.0 2.5</module>
```


```
</ptracker>
```

The specification can include an <input> and/or an <output> element with the standard capabilities, e.g., as for SENSORS or PINS. Similar to SENSORS, an input command addressed to the module is an ASCII line. If the first character of such a line is *T*, then the line describes the timing of the command in the next line, similar to SENSORS or PINS. Otherwise, that character can only be *C* (and any remaining characters in the line are ignored). This command clears (i.e., zeroes) the tracker effectively starting a new measurement at the current moment.

The <module> elements describe (intentionally) the current usage of the respective components (modules) in their different states. The meaning of those states depends on the component. The first state (and the first number) has a standard meaning representing the default (reset) state of the module. For example, value 0.3 at the CPU <module> can be interpreted as 0.3mA drawn by the board's CPU in the default idle state assumed after reset. Note that the interpretation of all those numbers is up to the user, and they have no implied standard meaning.

The second value at the CPU <module> represents the power down state (7.7uA). This component uses only two values.

For the RF (radio) component (the second module), we have 4 numbers describing, respectively, the current drain in the off state, with the receiver on, with the transmitter on, and with both the receiver and the transmitter simultaneously on (if it makes a difference). Generally, if the expected sequence of values is longer than the one provided, all the unspecified values are assumed to be the same as the last one. Thus, the last value for the radio <module> is redundant.

The storage module applies to EEPROM. The values respectively refer to the amount of current drawn by the module:

- in the off (or closed) state (e.g., after executing *ee_close*)
- in the on state (e.g., after *ee_open*) but otherwise idle
- while reading
- while writing
- while erasing
- while syncing

The amount of time needed for the module to remain in an active state (reading, writing, erasing, syncing) is determined from the timing parameters (page 26). If there are no timing parameters (for all or some operations), the corresponding current drain is not modeled (assumed to be zero).

For the sensors module, the second and possibly subsequent values refer to the current drain by the sensors or actuators (in the order of their numbers) while active. Note that sensors and actuators admit timing parameters (page 24). In the above example, the single “on” value (2.5mA) will apply to all sensors and actuators.

Not all modules have to be specified. A missing module uses no energy: its current drain is always zero regardless of the state.

If the output target of the power tracker is “device”, i.e., the updates regarding current drain are sent to a file, VUE^2 will be writing to that file a message after every change in the state of any of the tracked components. Such a message looks like this:

```
U T [E]: A V
```


where ***T*** is the time of the state change in seconds (from the beginning of run), ***E*** is the elapsed time of the current measurement (counted from the last moment the tracker was zeroed), ***A*** is the accumulated average current drain from the beginning of the measurement (since ***E*** seconds ago), and ***V*** is the current drain at the moment (after the state change). All four numbers are floating point (the last two may include exponents).

If the output target is "socket", a line in the above format is sent to the remote agent at 1 second intervals for as long as the connection is on.

4.12 Pre-initialization

By pre-initialization we understand assigning initial values to some node attributes in a way that is conceptually different from the normal (dynamic or static) initialization in the node program. Intentionally, it corresponds to burning some characteristic values into the node's flash memory, e.g., serial numbers, that make the node unconditionally distinguishable from other nodes.

Pre-initialized values are represented in the input data as <preinit> elements, which can be sub-elements of <node> or <default>. Here is an example:

```
<preinit tag="ESN" type="lword">0x8000ff01</preinit>
```

The mandatory *tag* attribute assigns a name to the preinit. The second (optional) *type* attribute determines the object type, which can be:

lword	a 32-bit long integer value (signed or unsigned)
word	a 16-bit integer value (signed or unsigned)
string	a character string

These indicators can be abbreviated down to the initial letter. If the type attribute is absent, it defaults to *word*.

The element's body contains the value, which should agree with the type specification. For types *lword* and *word*, the value should be a decimal number (possibly signed) or a hexadecimal value (beginning with *0x*). Regardless of its actual size, it will be truncated to the respective size of the object. For the *string* type, the body is simply viewed as a character string.

The way to reference a preinit in the praxis code is via this function:

```
IPointer preinit (const char *tag);
```

which returns the preinit value represented by the *tag*. The generic output type must be cast to the proper object type. This, however, only works for VUE^2. To make a preinit transparent from the viewpoint of both views of the praxis, you can use the *PREINIT* macro, e.g.,

```
lword NodeSerialNumber = (lword) PREINIT (0xBACADEAD, "ESN");
```

which accepts two arguments: the direct value to be used for the target system, and the symbolic preinit name to be used for VUE^2. When compiling for a real device, the macro will ignore the second argument and turn itself into a straightforward initializer for the variable. When compiling for VUE^2, it will ignore the first argument and become a call to *preinit*.

If the tag referenced by *preinit* is not found among the <preinit> elements associated with the current node, the function consults the preinits of <defaults>. If it is not found there, the

function returns zero, i.e., the object is initialized to zero (or to a NULL string pointer). This corresponds to leaving the object uninitialized in the real world.

5 Mobility drivers

The underlying SMURPH/SIDE vehicle allows the programmer to create arbitrarily elaborate mobility models as external SMURPH processes. VUE[2] provides an interface to external daemons that can modify coordinates of nodes by sending commands over sockets (Section 8.5). The same interface can be fed from files (or devices).

5.1 Declaring a mobility driver

A mobility driver accepting node positioning commands[8] is not a node module (like UART, LEDS or PINS), because its domain is not restricted to a single node: it may affect the positions of multiple nodes essentially at the same time. One such driver, capable of receiving connections from remote agents over sockets, is available all the time and need not be declared in the input data file. We shall call it the *external* driver. It is also possible to declare *local* mobility drivers, which can modify node positions according to a local description, i.e., included directly in the input data or read from local files. Such a driver accepts a subset of the commands available to a remote agent connected to the external driver

A local mobility driver is declared with the <roamer> element (see Illustration 2), which is a sub-element of <network> and looks like this:

```
<roamer>
    <input source="..."> ... </input>
</roamer>
```

where the input source specification is essentially the same as for PINS (Section 4.6). A <roamer> element with <input source="socket"> is ignored. Note that there is no <output> sub-element in <roamer>. While an external agent connected to a mobility driver over a socket can query the driver and receive feedback information (see Section 7.6), this functionality is not available to a local driver.

An arbitrary number of local mobility drivers can be declared in the same data file. Each of them can affect the position of an arbitrary number of nodes. Once a driver requests to re-position a node, any active positioning request regarding that node, possibly originating from another driver, is canceled and the new request takes over.

5.2 The commands

Essentially there are three types of commands accepted by a local mobility driver. These commands constitute a subset of requests available to external agents talking to a mobility driver over a socket. The simplest command simply assigns new coordinates to a given node and has this format:

```
M nn x y
```

where *nn* is a node number (according to the numeration of <node> elements), and *x* and *y* are the new planar coordinates. The operation amounts to teleporting the node to the new location. The move is instantaneous. Note that the coordinates cannot be negative.

Similar to PINS (page 21), it is possible to delay the interpretation of subsequent requests with a *T* command, e.g.,

[8]Strictly speaking, positioning commands apply to Transceivers rather than nodes.

```
T +0.01
```

will cause a 10 ms delay before interpreting the next request, while

```
T 3600
```

will halt the input until the model has been run for one hour since its startup.

With a sequence of appropriately timed (tiny) teleportations one can implement any movement, closely approaching smooth navigation along any curve at any (possibly variable) speed. It is tedious, however, to procure such data sets by hand. To simplify typical experiments, the mobility driver offers a standard random-way-point roaming model, which is invoked with the following command:

```
R nn x0 y0 x1 y1 smin smax pmin pmax duration
```

where all numbers except *nn* (the node number) are floating points. When such an request is accepted, the indicated node starts roaming an continues doing so for *duration* seconds according to the pattern prescribed by the remaining parameters. If *duration* is zero or negative, the roaming continues forever, or rather, until another repositioning request is issued to the node. In the meantime, the driver is free process other commands: the roaming is carried out in the background.

The first four floating point numbers, i.e., *x0, y0, x1, y1,* describe the rectangle bounding the node's roaming area. Note that *x0* must not be greater than *x1,* and *y0* must not be greater than *y1.* Besides, all coordinates must not be negative. The node will travel according to this algorithm:

1. A uniformly distributed random target location is generated within the bounding rectangle as well as a uniformly distributed random speed between the specified minimum (*smin*) and maximum (*smax*) in meters per second. The node moves at this speed towards the target location.

2. Upon reaching the target location, a uniformly distributed random pause time is generated between *pmin* and *pmax* seconds. The node rests for this much time and proceeds from 1.

This process continues for *duration* seconds, or indefinitely (if *duration* is zero or negative). It will be interrupted when the same or different mobility driver issues any re-positioning command to the node.

Taking advantage of the *string* source for the ROAMER, you can easily set up multiple random-way-point mobility scenarios for multiple nodes. For illustration, consider this declaration:

```
<roamer>
    <input source="string">
       R 0 [0.0 200.0 0.0 200.0] [3.5 10.0] [1.0 10.0] -1
       R 1 [0.0 200.0 0.0 200.0] [3.5 10.0] [1.0 10.0] -1
       T 1800.0
       R 5 [0.0 200.0 0.0 200.0] [3.5 10.0] [1.0 10.0] -1
       R 7 [0.0 200.0 0.0 200.0] [3.5 10.0] [1.0 10.0] 7200.0
       T +1800.0
       M 5 40.0 50.0
    </input>
</roamer>
```


Nodes 0 and 1 start roaming immediately at the beginning of the experiment. After 30 minutes, nodes 5 and 7 join them. Node 5 stops roaming after another 30 minutes when it is teleported to location (40.0, 50.0) and becomes stationary. Node 7 keeps moving for two hours before stopping (at some random location), while nodes 0 and 1 roam forever.

The above example suggests that the command syntax allows for non-numerical characters to be interspersed among the numbers (we used square braces for clarity). This only applies before floating point numbers and should be viewed as an undocumented feature that may be removed without warning. The maximum length of a single request line is 112 characters, which means that extravagance is not encouraged.

6 Panels

By default, the program (praxis) running at a node is automatically started as soon as the input data file is read and processed, at the very beginning of the model's execution. It is possible to have some nodes stopped initially by specifying *start="off"* in their <node> elements (see page 9). You may even have all nodes stopped initially (by putting start="off" as an attribute of <defaults>) and then start them later, e.g., at specific moments. Once started and running, a node can be stopped, started, and reset an arbitrary number of times. This process can be described in the input data (or a separate file); it can also be controlled by external agents.

6.1 Declaring panels

The respective data element, called <panel>, is illustrated in the sample data set on page 10. Its full format is similar to that of <roamer> (Section 5) and looks like this:

```
<panel>
    <input source="..."> ... </input>
</panel>
```

where the input source specification is essentially the same as for PINS (Section 4.6). A <panel> element with <input source="socket"> is ignored. Similar to <roamer>, there is no <output> sub-element in <panel>. While an external agent connected to a panel driver over a socket can control the status of nodes and receive feedback information (see Section 7.7), this functionality is not available to a local driver. An arbitrary number of <panel> elements can be present in the same data file. Each of them can affect the status of arbitrary nodes. For example, a node stopped by one panel can be restarted by another. Stopping a stopped node or starting a running node has no effect. Resetting a node always starts it up, regardless of whether it was stopped or running before the reset.

6.2 Commands

The simple set of commands understood by a panel consists of the timing request *T* – as for PINS (page 21) and three status change requests (in all cases, *nn* is the node number):

O *nn*	- to switch the node on
F *nn*	- to switch the node off
R *nn*	- to reset the node

For illustration consider the following sequence:

```
T 30.0
O 1
T +5.0
O 2
T +10.0
```

```
F 1
F 2
R 3
```

Thirty seconds after the beginning of execution node 1 will be switched on, then 5 seconds later, node 2 will be switched on, and after 10 more seconds, nodes 1 and 2 will be switched off and node 3 will be reset.

7 The agent protocol

A running VUE^2 model makes a socket available for connections from agents. Such agents may implement GUI to various station components (UART, LEDS, PINS, ROAMER) or provide an interface to OSS programs for the target praxis (executed on a real network). For example, an agent may implement an open ended UART driver (like a COM port under Windows) connected to a node running under VUE^2.

The standard port number of the socket opened by a VUE^2 model is 4443. It can be changed via the *port* attribute of <network> (see Section 4), e.g.,

```
<network nodes="48" port="5590">
```

which can be useful, e.g., if multiple VUE^2 models are run on the same system.

At present, VUE^2 implements eight functions (i.e., connection types) for agents. They are:

UART	for connecting to a node's UART
LEDS	for connecting to a node's LEDs module
PINS	for connecting to a node's pins module
CLOCK	for reading the virtual clock of the model
ROAMER	to affect the locations of nodes
PANEL	to control the on/off status of nodes
SENSORS	to receive values from virtual sensors and affect virtual actuators
LCDG	to send data to the virtual graphic LCD
PTRACKER	to receive updates regarding energy consumption by nodes

Except for ROAMER and PANEL, there can be multiple active instances of any connection type at any given moment – referring to the respective components of different nodes. For example, several UARTs belonging to different nodes can be interfaced to agents simultaneously.

In all cases, the interface is session-oriented, meaning that an agent does not connect to the model for a single inquiry, but sets up a session during which it will be involved in an exchange of potentially unlimited length. The connection can be broken by the agent at any time by simply disconnecting.

7.1 The handshake

A connection is initiated by the agent (client) connecting to the agent port of the VUE^2 model for a TCP (stream) session. Immediately after receiving a connection, the agent should send a polling sequence shown in Illustration 3.

It consists of 8 bytes interpreted as two short numbers and one long number in the network format, i.e., MSB first. The first number is a magic sequence used for a quick assessment of the request's sanity. The second short number, *code*, describes the requested service type. For a request related to a specific node (e.g., connection to a UART), the last four bytes specify the node number, according to the numeration of <node> elements in the input file (Section 4.3).

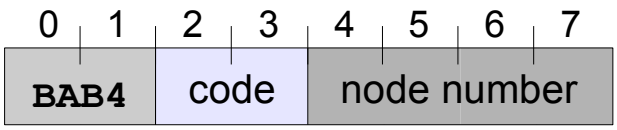

Illustration 3: Connection polling sequence

Having received a polling sequence, the VUE^2 program responds with four bytes (a 32-bit unsigned number in network format) interpreted as an acknowledgment. The least significant byte of that number indicates a success or failure. If its value is 129 (decimal), it means that the request has been accepted. The upper three bytes may return additional information to the connecting program. At present, only the UART module uses those bytes to pass the UART's bit rate as the number of bits per second divided by 100. Thus, an acknowledgment for UART connection has the format shown in Illustration 4. For the remaining modules, bytes 0-2 contain zeros.

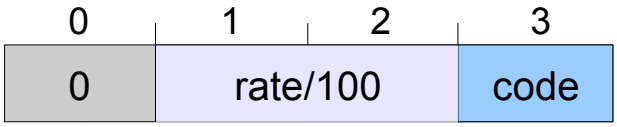

Illustration 4: UART connection response

If the acknowledgment code is different from 129, it means an error. Having sent the error code, the VUE^2 program immediately closes its end of the connection. Here is the list of possible error values:

0	wrong magic sequence
1	node number out of range
2	illegal (unimplemented) request code
3	some agent is already connected to this particular module
4	timeout; this is only sent if a complete 8-byte polling sequence does not arrive within 30 seconds from the moment of connection
5	the module (UART, PINS, LEDS, ...) is present at the node, but it has no socket interface
6	this code is sent when the program discovers that the other side has closed the connection, before closing its end; thus, it is unlikely to be ever received
7	the request sequence is too long
8	illegal request or query
128	the node has no such module (in response to a connection request)

7.2 UART protocol (request code 1)

This kind of request requires a valid node number. If the number is OK, and the indicated node is equipped with a UART module with socket interface, and no other agent is connected to that UART at the time, the request is accepted. Then, following the acknowledgment word sent by the VUE^2 program, the connection becomes entirely dedicated to the UART. This means that whatever the agent sends over the socket will appear as input on the UART, and whatever the praxis writes to the UART will be sent to the agent over the socket. This will continue until the session is torn down (closed) by the agent (or until the VUE^2 program terminates).

The format of the data sent/received by the UART conforms to the coding and type attributes associated with the UART in the input data (Section 4.4). In particular, if the input is “timed”, the data must be organized into packages preceded by the playback time. Normally, this kind of operation is not very useful for a socket connection, but it is available. Note that when the playback time is in the future, the input will be blocked until that time. Similarly, with “hex” encoding of the respective end, the data follows the hexadecimal format described in Section 4.4.

7.3 PINS protocol (request code 2)

This request requires a valid node number. If the number is OK, and the indicated node is equipped with a PINS module with socket interface, and no other agent is currently connected to the module, then the request is accepted. Following the acknowledgment word, the VUE^2 program immediately (without polling by the agent) sends an *N* message (indicating the number of pins – see page 22) followed by the full list of updates reflecting the current status of all pins. Each such a message takes a complete ASCII line of text, as described in Section 4.6, terminated with a single newline character. The agent may assume that the messages arrive in the order of pin numbers from zero up. Note that the total number of pins and the number of ADC-capable pins are included with every message.

No other message types ever arrive from the VUE^2 end. Whenever the status of a pin is changed by the praxis, a pertinent message is queued for transmission. Such a message always refers to a single pin.

The agent is able to affect the status of pins by sending messages over its end of the socket that look exactly like those described in Section 4.6. Such a message should be an ASCII string ending with a single newline character. With an on-line agent connection (as opposed to reading the pin status from a file), there is little demand on *T* requests, although they are not prohibited. Note that if such a message specifies a moment in the future, the input will be blocked until that time.

7.4 SENSORS protocol (request code 7)

This request requires a valid node number. If the number is OK, and the indicated node is equipped with a SENSORS module with socket interface, and no other agent is currently connected to the module, then the request is accepted. Following the acknowledgment word, the VUE^2 program immediately (without polling by the agent) sends an *N* message (indicating the number of actuators and sensors – see page 25) followed by the full list of updates reflecting the current values and ranges of all actuators and sensors. Each such a message takes a complete ASCII line of text, as described in Section 4.8, terminated with a single newline character.

No other message types ever arrive from the VUE^2 end. Whenever the value of a sensor/actuator is changed by the agent (in the first case) or by the praxis (in the second), a pertinent message is queued for transmission. Such a message always refers to a single sensor/actuator. Note that only the initial updates carry information about the ranges (Section 4.8).

The agent is able to affect the values of sensors (not actuators) by sending messages over its end of the socket that look exactly like those described in Section 4.8. Such a message should be an ASCII string ending with a single newline character.

7.5 LEDS protocol (request code 3)

This kind of connection works one way, i.e., following the initial 8-byte polling sequence, the agent never sends anything to the VUE^2 program. Immediately after the acknowledgment word, the VUE^2 program sends to the agent the initial configuration of LEDs as a message

described in Section 4.5. This is an ASCII message terminated with a single newline character. Then, a similar message is sent whenever the status of a LED changes.

Additionally, to be able to detect the disappearance of the agent, the program sends a dummy NOP message, which is simply a single newline character, every 10 seconds, unless there is a LED status change to report. Note that no NOP messages are sent when the LEDS module is interfaced to a file.

7.6 ROAMER protocol (request code 4)

This type of connection requires no node number (which is ignored, but must be present in the polling sequence). By following this protocol, the agent is able to carry out the operations described in Section 5.2, as well as receive feedback from VUE^2 regarding node positions. Only one remote ROAMER connection can be active at any time. Following the acknowledgment word sent by the VUE^2 program to the agent, the rest of the conversation is in ASCII, with lines terminated by (single) newline characters.

Immediately following the confirmation byte, the VUE^2 program awaits requests from the agent. The agent may query the program for the position of a given node with a command that consists of the letter *Q* followed by a single nonnegative node number, e.g.,

```
Q 49
```

In response, the program will send the following line:

```
P nn total x y name
```

starting with the letter *P* and consisting of two integer numbers in hexadecimal (*nn, total*), two floating point numbers (*x, y*) and a string (*name*). The first number is the node number and should be the same as the number sent in the query. The second number gives the total number of nodes in the network. The two floating point numbers are the node coordinates in meters. The string is the node's type name, i.e., the name of the SMURPH class representing the node. The latter is useful when the model consists of nodes of several types, as it allows the agent (with an appropriate set of queries) to recognize the complete configuration of the network. If the agent knows nothing, it can always safely issue a query for node 0. Then, having learned the total number of nodes, it can poll them all for position and type.

The agent may issue at will the commands described in Section 5.2. Also, it will receive an update whenever the position of a node changes, including changes incurred by the agent itself. Note that local mobility drivers may be changing things behind the scenes. VUE^2 tries to reduce the number of unnecessarily updates to avoid overflowing the connection with traffic under heavy mobility scenarios.

An update message begins with the letter *U* and looks like this:

```
U nn x y
```

where *nn* is the node number and *x* and *y* are its new coordinates.

Remember that node coordinates cannot be negative. Having received an invalid request, i.e., one specifying an illegal node number or a negative coordinate, the program sends error code 12 to the agent (a single byte) and closes the connection.

7.7 PANEL protocol (request code 5)

Similar to ROAMER, this type of connection requires no node number (which is ignored, but must be present in the polling sequence). By following this protocol, the agent is able to carry out the operations described in Section 6.2, as well as receive feedback regarding node status. Only one remote PANEL connection can be active at any time. Following the acknowledgment word sent by the VUE^2 program to the agent, the rest of the conversation is in ASCII, with lines terminated by (single) newline characters.

Immediately following the confirmation byte, the VUE^2 program awaits requests from the agent. In addition to the commands described in Section 6.2, one more (query) request is available to the remote agent, which looks like this:

Q *nn*

where *nn* is a node number. If *nn* is a legitimate node number, the program sends in response a line in the following format:

nn s total name

where *nn* is the node number specified in the original request, *s* is a single character O (the node is on) or F (the node is off), total is the total number of all nodes in the network (i.e., the limit for the range of legitimate node numbers), and *name* is the type name of the node, i.e., the name of the SMURPH class representing the node. Typically, an agent connecting to this service will begin its conversation with a query for node 0 (which is always safe) after which it will learn the total number of nodes.

In addition to being able to issue the commands described in Section 6.2, the agent will also receive an update whenever the status of a node changes (e.g., due to a panel described in the input data set – see Section 6), including changes incurred by the agent itself. Such an update message looks like a query response (see above), except that in includes only the first two items, i.e., the node number and its status. Note that when an active node is reset, its status does not change, so such an operation performed by a data-driven panel will be unnoticed by the agent.

Having received an invalid request/query, i.e., one specifying an illegal node number, the program sends error code 12 to the agent (a single byte) and closes the connection.

7.8 CLOCK protocol (request code 6)

With this simple protocol, the agent can receive time information from the VUE^2 program. Following the acknowledgment word, the program will be sending at second intervals the number of virtual seconds from the beginning of execution. This number arrives as a four-byte binary integer in the network format. Every message consists of exactly four bytes.

7.9 LCDG protocol (request code 8)

This is a one-way protocol with data being sent solely to the display. Following the acknowledgment word, the program is sending display updates consisting of binary short (16-bit) words in the network format. The most significant 4-bit nibble of a word determines the type of information carried in it:

0 means that the remaining 12 bits of the word contain the value of a pixel (4 bits per pixel: RGB, from most to least significant); the pixel is to be stored in the “current” position on the screen (as explained below)

1 means that remaining 12 bits of the word contain a command (as explained below)

8-15 i.e., the most significant bit being set, means that the word contains a repeat count for the last pixel; whatever last pixel value was received from the emulator, it must be repeated (*w* & 0x7fff) + 1 times (in addition to the original first appearance), where *w* is the word content

Here are the command codes:

0x300 (i.e., *w* = 0x1300); four words following this one contain the bounding rectangle of an area to be updated (filled) with subsequently arriving pixels; the rectangle is specified as the ending value of X, the starting value of Y, and the ending value of Y (the coordinate of the left upper corner of the display is 0, 0

0x400 end of update; this command means that a logically complete series of updates has been concluded and, for example, it now makes sense to present a new updated variant of the display to the user

0x100 switch the display on

0x200 switch the display off

In addition to defining a bounding rectangle, command 0x300 resets the current pixel position to the left upper corner of the rectangle, i.e., the starting X and Y coordinates. Whenever a new pixel (including auto replications) is stored, the X coordinate is advanced until it exceeds the ending coordinate, in which case it wraps back to the starting X coordinate, while the Y coordinate is incremented by 1. When the Y coordinate exceeds its ending value, it wraps back to the starting value, and so on.

7.10 PTRACKER protocol (request code 9)

Following the initial 8-byte polling sequence, the agent will be receiving from the simulator a message, as described in Section 4.11, every second or whenever the state of one of the monitored component changes, whichever comes first. This is an ASCII line terminated with a single newline character. When the agent wants to zero the tracker (i.e., start the measurement from scratch), it should send back a simple command consisting of the letter ***c*** followed by a newline.

8 UDAEMON

Here we briefly describe the functionality of udaemon, which is a Tk program implementing a rudimentary GUI agent for conversing with a VUE^2 model in execution. The program is started with an optional argument, which indicates the port number used by VUE^2 program. By default, this number is 4443 and agrees with the VUE^2 default (Section 7).

Note: the present version of udaemon requires Tcl/Tk version 8.5 or higher. The program may work correctly with older versions of Tcl/Tk, but the U-U feature (see below) will not work with version 8.4.

When started, the program opens a window that looks as shown in Illustration 5. This window is the launcher of five different interfaces, corresponding to the five protocols described in Section 7.

The text area labeled “Node Id” is used to insert the (decimal) node number, for those interfaces that require it. The “select” button, initially showing “UART (ascii)*,”* offers the menu

shown in Illustration 6. Having selected a given interface (and, if required, inserted a node number in the text area), you can click the *Connect* button to bring up the respective interface window.

The area below the buttons displays various textual messages (the log), which, in particular may include error messages. The area is scrollable and stores the last 1024 lines of the log (which practically always means everything).

Illustration 5: The root window of udaemon.

8.1 The UART interface

Note that the UART interface occurs in two versions on the menu: “ascii” and “hex.” These versions are independent of the “ascii/hex” coding discussed in Section 4.4. In this place, “hex” means hexadecimal display of the data arriving from the UART, as well as hexadecimal entry of the data to be sent to the UART, and is useful in those cases when that data are non-ASCII (e.g., the praxis is meant to communicate with some program using binary data, and we want to manually emulate the behavior of that program). A similar effect will be achieved when the UART coding is “hex” (and the interface mode at udaemon is “ascii”); however, in that case, the hexadecimal coding/decoding will be done by the VUE[2] program. Note that double “hex,” i.e., on both sides, will have a rather confusing effect of displaying in hexadecimal the character codes of hexadecimal digits (the input will be even more confusing).

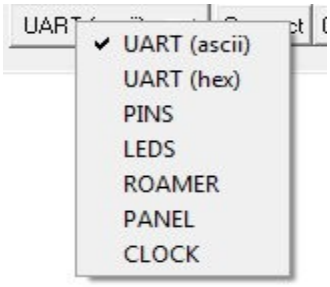

Illustration 6: The interface menu.

A UART connection produces a window shown in Illustration 7. It looks like a straightforward terminal emulator, with the text area at the bottom used for input. The upper area is resizeable and scrollable. It stores the last 1024 lines written to the UART.

By checking the “hex” box at the bottom, you can switch the terminal from “ascii” to “hex” and vice versa without reconnecting. The reason why, in addition to this box, the mode can also be selected from the root window is that when the UART window is open, some text may already be displayed on it (see the held option on page 18). The mode switch only affects the data to be displayed, not that already present on the screen.

A line typed into the bottom text area is sent to the VUE[2] program when you hit the *Enter* key. For “hex” mode of the UART terminal, this data should consist of pairs of hexadecimal digits optionally separated by spaces. Only the binary data represented by those digits will be sent to the VUE[2] program.

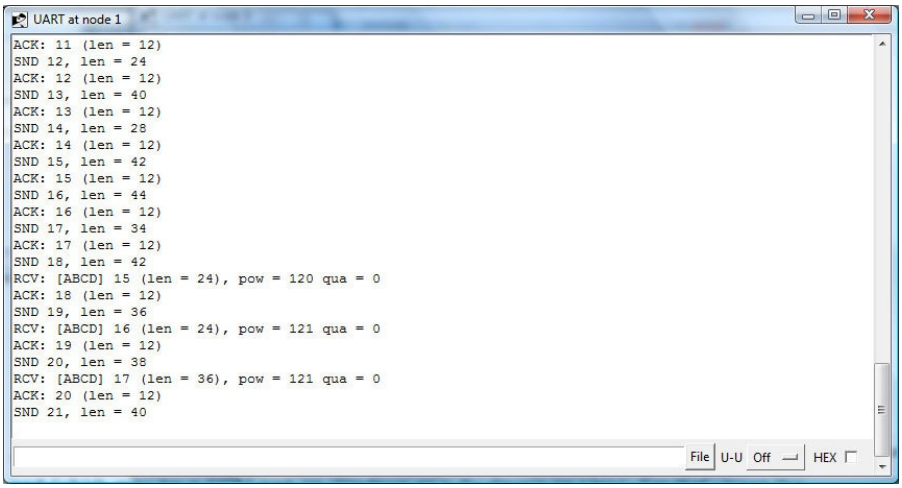

Illustration 7: A UART window.

The U-U button (whose default state is Off) allows you to interface the UART to a real serial port (which may be a COM port on Windows or a tty device on Unix). For that, press the button and select one of the devices available on the pop-up list. On Windows, entries looking like CNCB0, CNCB1, and so on, correspond to the B-ends of COM-to-COM port pairs emulating null modems. The requisite driver (for Windows) can be downloaded from http://com0com.sourceforge.net/.[9] This way, you can have real-life OSS programs talk to PicOS praxes executed under VUEE (with the OSS program talking to the A-end of the respective pair).

On Linux, the same feature can be implemented using pseudo-ttys, i.e., devices named /dev/pts/*n*, where *n* is a number. In directory PICOS/Linux (of the PicOS package, not VUEE), you will find a file named *nullmodem.c*, which compiles trivially, e.g.,

```
gcc -o nm nullmodem.c
```

When you run the resulting executable, it will create two pseudo-ttys, show them to you, e.g.,

[9] This package is also available from http://www.olsonet.com/REPO/com0com-2.2.0.0-i386-fre.zip (32-bit version) and http://www.olsonet.com/REPO/com0com-2.2.0.0-x64-fre.zip (64-bit version). For best effects, having installed the driver and set up a pair of ports, say CNCA0 and CNCB0 (which are created by default), invoke the *setupg* utility and select “emulate baud rate” and “enable buffer overrun” for both ends.

```
TTY name: /dev/pts/7
TTY name: /dev/pts/6
```

and, from that point on, act as a forwarder between them. One of those pseudo-ttys should then be selected from the U-U menu of the UART window, while the other can be opened by the OSS program. The formal baud rates used at the two ends don't have to match.

When a virtual UART of udaemon is connected to a real serial device, the UART window mirrors the exchange between the node and the program attached to the other end of the real port. The input field also works in that mode (your entries will be interleaved with any data arriving from the real UART), as does the HEX checkbox.

The File button appearing to the left of U-U (and to the right of the input field) can be used to open a file where the data written to the UART will be stored. Once such a file has been opened, the button label changes to “Stop”. When you press it, you will close the file and stop the operation of saving there the UART data.

8.2 The PINS interface

Similar to UART, this interface requires a valid node number. Following a successful connection, a window like the one shown in Illustration 8 pops up on the screen.

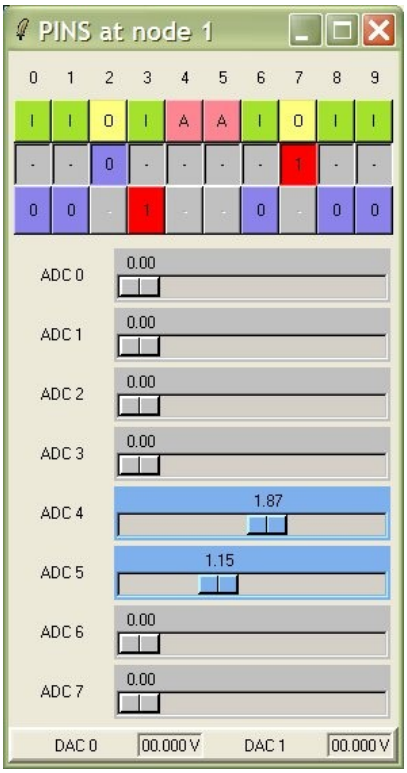

Illustration 8: A PINS window.

The three rows of boxes at the top reflect the current status of the pins, with one column corresponding to one pin. Only the bottom row is clickable, and only if the letter in the upmost box of the column is *I*, which means that the pin has been set by the praxis to “input.” For such a pin, clicking on the box in the bottom row toggles the binary value of the

pin. In Illustration 8, pins 0, 1, 3, 6, 8, 9 are input, pins 2 and 7 are output (letter *O*), and pins 4 and 5 are set for analog input (letter *A*). The full collection of characters that can appear in a bottom-row box is:

I	the pin is set for digital input
O	the pin is set for digital output
A	the pin is set for analog input
P	the pin is a pulse counter
N	the pin is a notifier
D	the pin is a DAC output
–	the pin is unused (its status field in the input data set is 0, see page 20)

The middle row shows the output value of those pins that have been set by the praxis as "output." For any other pin type, the corresponding square contains a dash. Those squares in the bottom row that do not represent input pins are disabled, i.e., clicking on them triggers no action.

A right click on a clickable pixel square has the effect of two consecutive clicks separated by a 300 ms interval. The role of this feature is to emulate pressing a button (Section 4.7).

Each ADC-capable pin has a slide widget that can be used to set the voltage on that pin. Only those pins that have been currently selected for the ADC function have their slides enabled (pins 4 and 5 in Illustration 8). The value above the slide knob tells the current voltage on the pin. Finally, the DAC-capable pins have voltage display areas at the bottom. No such pins appear to be available in Illustration 8.

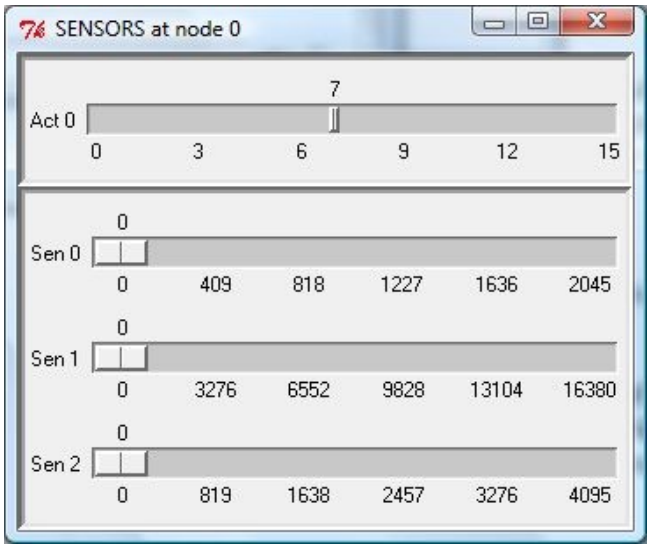

Illustration 9: A SENSORS window.

Note that the praxis can dynamically redefine pins, and, for those that are output or DAC, set their values. All such changes are immediately reflected in the window. You can trigger changes in those pin values that are currently available for digital input or ADC – by clicking on a box in the bottom row or adjusting the respective slide. Such changes are immediately conveyed to the VUE^2 program. Slide adjustments are sent incrementally, meaning that a slow movement of the slide knob will result in several updates sent to the VUE^2 program, reflecting the progress in adjustment.

8.3 The SENSORS interface

Similar to PINS, this interface requires a valid node number. Following a successful connection, a window like the one shown in Illustration 9 appears on the screen. The upper section displays the values of the node's actuators. There is no way to affect an actuator value from the window.
Every sensor defined at the node has a slider in the lower portion of the window. By adjusting the sliders, you can modify the sensor values. An update is sent when the mouse is released.

8.4 The LEDS interface

This interface is very simple as it involves no user input. The displayed window shows one circle for each of the LEDs defined in the module, with the LEDs numbered from left to right. Color gray means that the LED is off, otherwise, it is on. The first five LEDs show the colors: red, green, yellow, orange and blue when lit. If there are more than five LEDs, the ones numbered 5 and up are all red.

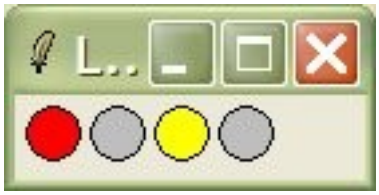

Illustration 10: A LEDS window.

8.5 The ROAMER interface

This interface takes no node number. Illustration 11 shows a sample window displayed in response to a ROAMER connection, for a network consisting of 6 nodes of the same type. The numbers in the lower right corner show the dimensions of the area covered by the window.

If the window is resized, udaemon will re-fit the nodes to the new size, i.e., by extending the window, you do not necessarily make it cover a larger area, but simply magnify the picture. It isn't difficult to guess that by dragging a node, you will move it to a new location. If the dragging is smooth, multiple requests will be sent to the VUE^2 program, such that the movement will appear incremental to the model. Note that the present version of udaemon has no interface for specifying random-way-point mobility patterns (see page 32).

Moving to the right and up is unrestricted. If you move a node beyond the window boundary, the window will be renormalized to the new network geometry. This renormalization does not affect the window's size or shape on the screen, but the assumed dimensions of the edges. Moving to the left and down is limited by the coordinates <0,0>. Udaemon will not let you move a node below these coordinates.

To see the exact position of a node, just move the mouse over it. The coordinates will be displayed in the lower left corner of the window along with the node's type name, which coincides with the name of the node's class. If there are multiple node types in the model, they will receive different colors on the screen – up to six colors: yellow, blue, orange, red, green, gray. If there are more than six node types, all the remaining types will be gray.

Illustration 11: A ROAMER window.

There is a way to see the distance between a selected pair of nodes. Click on the first node without dragging it, then move the mouse over the second node and click. A dashed line connecting the nodes will show up for 2 seconds with a number in the middle showing the distance between the nodes in meters.

8.6 The PANEL interface

This interface takes no node number. Illustration 12 Shows a sample panel window. Immediately after the request is issued, the window only displays the status of node 0. More nodes can be added to the window by entering the node number in the text area at the bottom and pressing the *Add* button. Nodes can also be removed from the panel by pressing the blue *Delete* button on the right.

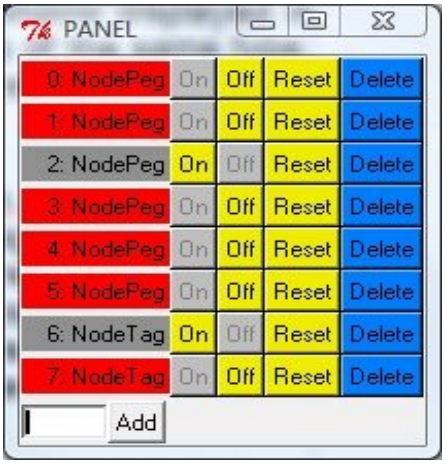

Illustration 12: A panel window.

As shown in Illustration 12, nodes 0, 1, 3-5, and 7 are on, while nodes 2 and 6 are off (halted). The on status of a node is indicated by the red background color of the node's label. For the three status change buttons, color yellow means that the button is enabled. Note that Reset is always enabled: resetting an off node will also start it up and make active.

8.7 TIMER interface

This is a very simple interface that needs no node number. It opens a window that shows the emulated time of the model in seconds, assuming that the execution started at time 0.

8.8 LCDG interface

This is a simple passive (reception only) interface presenting a 130x130 pixel display looking as shown in Illustration 13.

Illustration 13: LCDG model

8.9 PTRACKER interface

This is a single window looking like the one in Illustration 14.

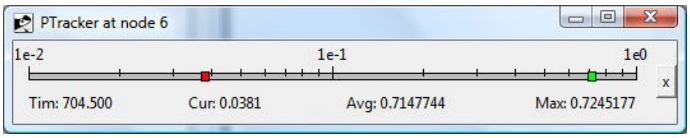

Illustration 14: PTRACKER window

The red bar shows the current current drain (note that the scale is logarithmic), while the green bar shows the accumulated average. The small button on the right is used to zero the power tracker.

8.10 Pre-configuring windows

Sometimes, e.g., for a demonstration, you may want to predefine a (possibly complex) initial configuration of windows declaring their positions on the screen and selecting some of them to be brought up automatically when udaemon starts. Such a configuration can be described in a file using simple XML-like declarations. The default expected name of that file is *geometry.xml*. If a file with this name is present in the current directory when the program is invoked, it will use that file automatically. A different file can be specified explicitly with the *-G* option, e.g.,

```
udaemon -G /home/pawel/MyStuff/windows.def
```

The configuration file allows you to specify:

- Positions of windows on the screen. For node-relative windows (ones that may occur in multiple versions parameterized by the node number) you can declare different positions for different nodes or groups or nodes.
- Which windows should be displayed initially. Node-relative windows are selected individually on a per-node basis.
- A background image for the ROAMER window.

Here are the contents of an example configuration file:

```
<geometry>
      <root> =(10,10) </root>
      <roamer display="yes">  =(100,230) </roamer>
      <uart> 0,1,6-9,15=(350,250) </uart>
      <uart display="yes"> 2=(720,410) </uart>
      <clock> =(-170,-45) </clock>
      <panel display="yes"></panel>
</geometry>
```

All the declarations appear as sub-elements of the single <geometry> element. Elements corresponding to the particular window types are named after the respective windows (e.g., as seen on the root menu) in all lower-case. Here is the full list of legitimate names: *uart*, *clock*, *leds*, *lcdg*, *pins*, *sensors*, *panel*, *ptracker*, *roamer*, and *root*, with the last one representing the root window (see Illustration 5).

The minimum acceptable syntax for a location description, which comes as the text body of the respective XML element, consists of = followed by two numbers in parentheses (white spaces are ignored). The first number specifies the x-coordinate (pixel number) of the left edge of the window, and the second number gives the y-coordinate (pixel number) of the upper edge. A negative number (like in the <clock> element in the above example) represents a negative offset (in pixels) from the maximum coordinate, i.e., from the right/lower boundary of the screen.

If = is preceded by a list of numbers, then those numbers refer to the nodes to which the description should apply (this only makes sense for node-relative windows). For example, the first <uart> element from the above example will be applied to the UART windows for nodes 0, 1, 6, 7, 8, 9, and 15. All those windows will be displayed in the same location, which probably makes little sense, unless, for some reason, only one of them is displayed at a time.

Wherever it makes sense, a location description may consist of multiple entries in the form "*list of nodes = (coordinates)*" separated by spaces or commas, e.g.,

```
0 = (40, 200), 1-4,7 = (70, 200), = (300, 800)
```

Note that the last entry in the above sequence covers all the cases that are not mentioned with the preceding entries. This is implied by the fact that the list of entries is examined from the front, and the first entry that matches the window's node number is applied.

Multiple location descriptors can also be split over multiple elements with the same name, e.g., this definition:

```
<leds> 0=(10,200) 1=(120,200) 2=(230,200) 3=(340,200) </leds>
```

is equivalent to:

```
<leds> 0 = ( 10,200) </leds>
<leds> 1 = (120,200) </leds>
<leds> 2 = (230,200) </leds>
<leds> 3 = (340,200) </leds>
```

If the *display* attribute of a window element is *yes*, the element describes one or more windows to be brought up automatically when the program starts. If such an element refers to a node-relative window, then 1) it must also specify a location, and 2) the node selector of that specification (the string preceding =) must consist of a single number. The element may include more than one location specification, but only those specifications for which the node selector is a single number will select the windows to be initially displayed. For example,

```
<leds display="yes">4=(10,200) 5-7=(10,400) 8=(10,700)</leds>
```

will bring up LEDS windows for nodes 4 and 8. The location specification for nodes 5, 6, 7 will be applied when those windows are requested manually; however, they will not be displayed automatically when udaemon starts. For the windows that occur in single versions the position specification is optional, e.g.,

```
<clock display="yes"></clock>
```

will work as intended, i.e., the window will be brought up upon startup (but its location is not determined and left to the discretion of the window manager).

Note that setting the *display* attribute for the *root* window is redundant (that window is always displayed as soon as udaemon starts).

The <roamer> element allows for two additional (and optional) attributes, e.g.,

```
<roamer image="imgfile.jpg" width="1200.0">=(10,20)</roamer>
```

which can be used to specify a background image for the ROAMER's canvas. That file must be in one of the formats acceptable by Tk (the Img package), e.g., *jpeg* files are OK. The path to the image file will be interpreted from the directory in which udaemon has been called. Attribute *width* associates a width with that file, i.e., specifies the extent of the image's width in the geographical units (meters) used by VUEE to interpret node coordinates. The corresponding height is derived from the ratio of the image's height to its width.

With a background image defined, the ROAMER window becomes inflexible and cannot be resized. The program assumes that the network area covers a rectangle of the specified width (and the derived height) starting at point (0,0). Thus, you should make sure that node coordinates specified in the respective VUEE data file (see page 15) fall into that area. Some narrow margins within the image area (stripes at the bottom, top, left, and right) are reserved and cannot be used for node coordinates (they roughly correspond to 1/2 of the node image size). If a node coordinate falls outside the network rectangle, udaemon will complain (with a pertinent message presented in the root window) and display the node at the window's boundary (such that all nodes are always accounted for). In contrast to the standard version of the window, nodes cannot be moved (with the mouse) outside the boundary of the fixed network rectangle. Note that they can still be moved by the simulator (if it runs a mobility model), in which case their images will be pushed to the window's boundary.

Instead of *width*, you can specify *height* (in that case the width will be derived). As long as the image attribute is present, at least one of them (*width*, *height*) is required. You can also specify both *width* and *height*, which (if they don't agree with the image dimensions) will override the image scaling. Note that the text field of an element specifying a background image for the ROAMER can be empty, i.e., you are not obliged to specify a predefined location for the window.