Olsonet Communications

VUE^2

Version 0.5
February 2007

1 Introduction

VUE^2 or (VUEE) which stands for Virtual Underlay Emulation 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. 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 the first draft of VUE^2's description, including its interfaces needed by a PicOS developer to maintain compatibility with VUE^2 (such that the PicOS praxis can be directly run under VUE^2). As the system is being developed, it is likely to change, not only in the course of its natural evolution, but also in response to new features being added to PicOS (which will be mirrored in VUE^2). At present, VUE^2 captures a significant subset of all PicOS features, including drivers for RF modules, external memories and UARTs. Most importantly, it provides for an easy expression of network models and their virtual deployment. This means that PicOS praxes 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), VUE^2 also facilitates the development of various agents that have to be interfaced to that praxis to make the overall system complete.

This document will not be comprehended without the following documents by Olsonet Communications: *PicOS, version 2.01* [or higher], *VNETI: Virtual NETwork Interface, version 2.01* [or higher], *SIDE/SMURPH: a Modeling Environment for Reactive Telecommunication Systems, version 2.8* [or higher].

2 Similiarities 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 respective 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 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, as 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 intentionally different. The actual execution time of a SMURPH thread is essentially irrelevant (unless it renders the 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, an extensive period of CPU activities can be modeled in SMURPH by appropriately (and explicitly) delaying certain state transitions.[1]

Consequently, at the first level of approximation (and by default), 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 porting a PicOS praxis to 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 impression 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 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 (or praxes); also interfacing and parameterizing 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 (re)organized for execution under VUE^2. Notably, with the right structure (explained in the next section), the praxis may exist in a single version, which is shared between PicOS and VUE^2. Most of the code is directly shared. There is no need to convert it from PicOS to VUE^2 or vice versa. This advantage of emulation in VUE^2 cannot be overstated: **the emulated execution deals with the real code.**

3 The structure of VUE^2 compiler

The VUE^2 compiler, i.e., the program that turns a collection of PicOS praxes into an executable model of the network, is in fact the *mks* program of SMURPH, i.e., the compiler of SMURPH models into executable programs. Here is the list of components needed to set

[1]We are 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.

up the complete platform. These components can be configured under Cygwin (Windows) or directly under a UNIX system, e.g., Linux.

1. The “standard” 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 certain files that are directly used by VUE^2.

3. VUE^2 specific environment consisting of 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 *udaemon*. This is a Tk script implementing a rudimentary interface (GUI) to VUE^2 models.

The exact way of setting up SMURPH and PicOS is described elsewhere. Here we shall focus on the VUE^2-specific issues. For illustration, suppose that the three components have been unpacked at the same level (e.g., in the user's home directory). They fill three separate source trees rooted at directories VUEE, PICOS and SIDE.

3.1 Notes on configuring SMURPH/SIDE with VUE^2

When setting up SMURPH/SIDE (according to the instructions in the manual), you have to specify, when requested by *maker*, the proper location of the *include libraries* needed by VUE^2. This location is VUEE/PICOS. Thus, the relevant fragment of your conversation with *maker* when installing the package should look like this:

```
Give me the path to the include library (absolute or relative to
your home directory). The default path is:

        /home/.../SIDE/Examples/IncLib

which is equivalent to: ../Examples/IncLib
VUEE/PICOS
```

where the last line (in italics) is your response (it assumes that VUE^2 has been unpacked in home directory). Note that default proposed by maker is not appropriate (it is geared for the standard installation of SMURPH as a network simulator).

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.

It makes sense to use a non-default name for the *mks* program generated by *maker*, i.e., the actual compiler of VUE^2 models. The recommended name is *vuee* or *vue2*. When called under one of these names, the compiler automatically forces –L and –W, i.e., selects the visualization mode of SMURPH and disables the (redundant) models of wired channels. Note that all VUE^2 models require –W to compile.

It is possible to have a single installation of 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 selecting the default (or other) *include library* and assigning the standard (or different) name to the *mks* compiler. Depending on which compiler version is invoked, the corresponding *include library* will be referenced and the respective set of options will be applied.

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. Having unpacked the directory, move to VUEE/PICOS and execute *./mklinks* in that directory. That will set up a few links to PICOS files needed by VUE^2 and will only work if that directory occurs at the same level as VUEE. If this is inconvenient for whatever reason, you can edit the *mklinks* script specifying the proper path to PICOS.

Certain files in the PICOS tree (those linked to from VUEE/PICOS) are directly compiled as components of VUE^2 models. In particular, *tcv.c* (VNETI) is one of those files. This greatly simplifies the compatibility issues for networked praxes.

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. Having installed everything, you can compile those PICOS praxes that have been made VUE^2 compliant by moving to the praxis directory and executing *vuee* (or whatever name you assigned to the VUE^2 SMURPH compiler). The outcome of this compilation will be an executable file named *side* (*side.exe* under Cygwin). By running this file (with a suitable data file), you will execute the model.

A VUE^2 compliant praxis can be compiled by both, the VUE^2 compiler as well as the PicOS compiler. In the latter case, you simply execute *mkmk boardname* in the praxis directory followed by *make*. A certain extension has been added to the PicOS compiler to accommodate multiple praxes in the same directory. This is needed in situations when a single VUE^2 model must accommodate multiple praxes (different node types). Such praxes being related, it makes sense to keep them in the same directory also from the viewpoint of PicOS.

Traditionally, in the single-praxis-per-directory case, *mkmk* seeks a file named *app.c*, which is treated as the root file of the praxis. The configuration of its include files + the board options + the possible extra options specified in *options.sys* in the praxis directory determine the constituents of the target Image file to be loaded into the microcontroller. If there is no file named *app.c*, but there is another file whose name looks like *app_xxx.c*, *mkmk* assumes that this is the praxis root and compiles it instead. The resulting image files will be called *Image_xxx* (the ELF version) and *Image_xxx.a43* (the Intel hex version). If multiple files named *app_...* are present, e.g., *app_one.c* and *app_two.c*, *mkmk* will assume that there are multiple praxes in the directory and generate a *Makefile* to compile them all into separate images (*Image_one* and *Image_two* in this case). A VUE^2 compliant configuration of multiple praxes will follow this convention to make those praxes discernible by *mkmk* (PicOS) while keeping them unified for the purpose of their combined model in VUE^2.

One more feature is the treatment of source files whose names end with the suffix *.cc*. As we shall see later (step 7 on page 16), this suffix is needed for a file to be compiled by SMURPH. If a file with this suffix becomes needed by the praxis when compiled under PicOS, the compiler copies it to a similarly named file with the suffix *.c* and compiles it from there. This also applies to praxis roots, i.e.,files named *app_xxx.cc.*

4 Converting PicOS praxes to a VUE^2 compliant format

We start from the list of general guidelines. On top of those guidelines are some further restrictions mostly resulting from the fact that not all PicOS operations are currently modeled in VUE^2. However, the ones that are modeled are sufficient for practically all our present praxes. Besides, the list of deficiencies will shrink as the modeled components are extended to accommodate new praxes. It is postulated that all new praxes be programmed in the

VUE[2] compliant style from the beginning. Should a need arise to extend the assortment of VUE[2] tools, we will be quick to respond to such requests.

4.1 General guidelines and comments

These guidelines can be viewed as prerequisites for an easy (mostly automatic) conversion of an existing praxis that was programmed with no VUE[2] compatibility in mind. This is to say, the first step in such a conversion would be to make sure that the praxis code follows these guidelines.

1. Construct your threads using these operations: *thread, strand, endthread, endstrand,* instead of *process* and *endprocess*. Similarly, use *runthread, runstrand* instead of *fork*.

2. Do not use *static* declarations within a process function. Such declarations are intended to express "permanent' local variables of processes. Instead, move them all to the global scope and use discerning prefixes or suffixes in case of conflicts. Note that keeping all *de facto* global variables in one place helps avoid memory fragmentation on misalignment in a PicOS incarnation of the praxis.

3. Do not use *entry* constructs with numerals or expressions, e.g.,

   ```
   entry (RS_CMND+1)
   entry (0)
   ```

   Make sure that all state arguments in those operations that accept state arguments are simple constants representing state names. This means that all states must be assigned distinct symbolic names.

4. Identify those global variables that must be statically initialized from those that need not. It may make sense to group the two classes into two contiguous chunks. There is no penalty for initializing some of the latter (say with zero), other that such an initialization will have to be explicit at some point.

The primary source of some ugly complications related to points 2 and 4 is the fact that all *de facto* global variables in the PicOS praxis must be turned into attributes of a Node object in the VUE[2] model. This is because what looks like a complete program from the viewpoint of PicOS becomes merely a set of methods run by one node in the VUE[2] program.

The idea behind the PicOS to VUE[2] conversion is straightforward: PicOS threads are directly transformed into SMURPH processes run at stations representing network nodes. There are two types of variables referenced by such a process: local (automatic) variables of the process (i.e., ones allocated on the stack that do not survive state transitions) and global (permanent) variables. The automatic variables can be mostly left alone, as they appeared in the original PicOS code. The global variables must be turned into attributes of a SMURPH station representing the given node. It makes no difference whether they are *static* in a thread or explicitly global. C++ cannot tell the difference between the two kinds. Moreover, as the code of a PicOS thread will (quite mechanically) become the code method of a SMURPH process, all *static* declaration must be removed from it, as their interpretation for a C++ method would be quite different from the one intended.

Let us consider a sample praxis that has been written with no regard for VUE[2]. We shall use it to illustrate the conversion process. You can find it in PICOS/Apps/VUE/ILLUSTRATION. Here is the complete code of this praxis as it once appeared in the app.c file (see app.c_original in ILLUSTRATION):

```
#include "sysio.h"
```

```
#include "tcvphys.h"
#include "ser.h"
#include "serf.h"
#include "form.h"
#include "phys_dm2200.h"
#include "plug_null.h"

#define      MAX_PACKET_LENGTH   60
#define      IBUF_LENGTH         82

int    sfd;
word   Count;

void show (word st, address pkt) {

       ser_outf (st, "RCV: %d [%s] pow = %d qua = %d\r\n",
             Count++,
             (char*)(pkt + 1),
             ((byte*)pkt) [tcv_left (pkt) - 1],
             ((byte*)pkt) [tcv_left (pkt) - 2]
       );
}

#define      RC_TRY       0
#define      RC_SHOW      1

thread (receiver)
  static address packet;
  entry (RC_TRY)
       packet = tcv_rnp (RC_TRY, sfd);
  entry (RC_SHOW)
       show (RC_SHOW, packet);
       tcv_endp (packet);
       proceed (RC_TRY);
endthread

word plen (char *str) {

       word k;
       if ((k = strlen (str)) > MAX_PACKET_LENGTH - 5) {
             str [MAX_PACKET_LENGTH - 4] = '\0';
             k = MAX_PACKET_LENGTH – 5;
       }

       return (k + 6) & 0xfe;
}

#define      SN_SEND      0

strand (sender, char)
  address packet;
  entry (SN_SEND)
       packet = tcv_wnp (SN_SEND, sfd, plen (data));
       packet [0] = 0;
       strcpy ((char*)(packet + 1), data);
       tcv_endp (packet);
       finish;
endstrand

#define      RS_INIT      0
#define      RS_RCMD_M    1
#define      RS_RCMD      2
#define      RS_RCMD_E    3
```

```
#define      RS_XMIT      4

thread (root)
  static char *ibuf;
  entry (RS_INIT)
       ibuf = (char*) umalloc (IBUF_LENGTH);
       phys_dm2200 (0, MAX_PACKET_LENGTH);
       tcv_plug (0, &plug_null);
       sfd = tcv_open (NONE, 0, 0);
       tcv_control (sfd, PHYSOPT_TXON, NULL);
       tcv_control (sfd, PHYSOPT_RXON, NULL);
       if (sfd < 0) {
              diag ("Cannot open tcv interface");
              halt ();
       }
       runthread (receiver);
  entry (RS_RCMD_M)
       ser_out (RS_RCMD_M,
              "\r\nRF S-R example\r\n"
              "Command:\r\n"
              "s string  -> send the string in a packet\r\n"
       );
  entry (RS_RCMD)
       ser_in (RS_RCMD, ibuf, IBUF_LENGTH-1);
       if (ibuf [0] == 's')
              proceed (RS_XMIT);
  entry (RS_RCMD_E)
       ser_out (RS_RCMD_E, "Illegal command\r\n");
       proceed (RS_RCMD_M);
  entry (RS_XMIT)
       runstrand (sender, ibuf + 1);
       proceed (RS_RCMD);
endthread
```

Illustration 1: A sample PicOS praxis.

Note that certain steps to ensure its VUE[2] compliance have already been taken (operations *thread*, *strand*, and so on). In the first step, we should make sure that there are no implicitly global variables declared in the threads, like the ones highlighted in the above code. Consequently, we shall remove them from the threads and put them in front, together with other global declarations:

```
...
int    sfd;
word   Count;
...
address r_packet;   // Used to be static at receiver
char    *ibuf;      // Used to be static at root
...
thread (receiver)
  entry (RC_TRY)
       r_packet = tcv_rnp (RC_TRY, sfd);
  entry (RC_SHOW)
       show (RC_SHOW, r_packet);
       tcv_endp (r_packet);
       proceed (RC_TRY);
endthread
...
thread (root)
  entry (RS_INIT)
...
```

Illustration 2: Eliminating static declarations from threads.

In some cases, a static variable removed from a thread header may have to be renamed to avoid overlaps with similar variables declared in other threads. As *packet* seems to be a rather popular name for a variable, we have done so in the above example. Note that the automatic declaration of *packet* in the sender strand is left as it was.

4.2 Beginning the conversion

In essence, a praxis looking like the one in Illustration 1, with the modifications shown in Illustration 2, can be converted to VUE^2 mechanically. Here is one way to do it described step-by-step. While presenting those steps, we will be digressing into detailed explanations of the underlying issues and their solutions. Note that what truly matters is the spirit of the conversion, i.e., once you realize what needs to be done, your favorite way of rendering a praxis VUE^2 compliant may differ in details. In particular, the exact names and structure of the files introduced in this process need not strictly follow our prescription.

Step 1: Create a file named *threadhdrs.h* in the praxis directory. The role of this file will be to assure the formal compatibility of threads for each of the two platforms. It will be included from *app.c* (and possibly other praxis files that define threads). In our case, this file may look as follows:

```
#ifndef __praxis_threadhdrs_h__
#define __praxis_threadhdrs_h__

#ifdef __SMURPH__

process THREADNAME (receiver) (Node) {
	address r_packet;
	states { RC_TRY, RC_SHOW };
	perform;
};

process THREADNAME (sender) (Node) {
	char *data;
	states { SN_SEND };
	void setup (char *d) { data = d; };
	perform;
};

process THREADNAME (root) (Node) {
	char *ibuf;
	states { RS_INIT, RS_RCMD_M, RS_RCMD, RS_RCMD_E,RS_XMIT };
	perform;
};

#else /* PicOS */

// ========================================
#define	RC_TRY		0
#define	RC_SHOW		1
address r_packet;
// ========================================
#define	SN_SEND		0
// ========================================
#define	RS_INIT		0
#define	RS_RCMD_M	1
#define	RS_RCMD		2
#define	RS_RCMD_E	3
#define	RS_XMIT		4
char *ibuf;
```

```
#endif /* SMURPH or PICOS */
#endif
```

Illustration 3: The contents of *threadhdrs.h*.

The idea is that with this file in front of a thread code, that code will be interpreted correctly by PicOS as well as SMURPH. The constant *__SMURPH__* can be used as a general and authoritative way of telling whether a piece of code is being compiled by VUE[2] or by the PicOS compiler.

The sequence of *process* statements turns the praxis threads into SMURPH processes. The actual name of the process (as seen by SMURPH), may slightly differ from the PicOS name. The responsibility of the *THREADNAME* macro is to properly adjust this name in those circumstances when an adjustment is required. To see the problem, suppose that different PicOS praxes are combined into a single VUE[2] model. When those praxes are handled by the PicOS compiler, they are completely independent, in the sense that the name spaces of their threads are disjoint and do not interfere. However, when the same praxes are combined into a single SMURPH program, we have to eliminate potential name overlaps. For example, two praxes may use the name *sender* for their private variants of some thread.[2] This issue only becomes relevant when the model consists of multiple praxes (we shall return to it later). In the present case, we could simply remove the macro and, instead of:

```
process THREADNAME (receiver) (Node) {
```

simply say:

```
process receiver (Node) {
```

However, obeying this convention in all praxes makes their VUE[2] incarnations independent and easily re-combinable into compound models.

Those variables that, according to the PicOS view, are local to threads but permanent (i.e., were declared as *static* within a thread scope in the very first version of the praxis code) can be safely turned into process attributes in the SMURPH version (as shown in Illustration 3). This is not the only solution: they can also be treated as straightforward global variables, as they *de facto* are in the PicOS incarnation of the code. From the viewpoint of SMURPH, their interpretation as process attributes may be more methodologically correct. Whatever variables you decide to view this way, their global declarations must appear in the PicOS part of *threadhdrs.h* (the highlighted lines). The idea is that the file takes care of those things that are formally specific to the threads. Thus, for example, as *r_packet* is a process attribute in the SMURPH incarnation of the *receiver* thread, it should be mentioned as *global* in the PicOS section of the file.

Note that, as the *sender* process is a strand, i.e., it takes a private data object, its SMURPH variant must define the *data* attribute, whose pointer type must agree with the type specified as the second argument of the respective strand declaration (page 7). This attribute must be initialized from the setup argument when the process is created.

Step 2: Create two files named *node.h* and *node.cc*. The first of them should look like this:

```
#ifndef __praxis_node_h__
#define __praxis_node_h__

#include "board.h"
```

[2]Note that processes in SMURPH are not station methods, and the names of their types are global.

```
#include "chan_shadow.h"
#include "plug_null.h"

station Node : NNode {

#include "attribs.h"
#include "starter.h"

       void setup (data_no_t*);
       void _da (reset) ();
       void init ();
};
#endif
```

Illustration 4: The contents of *node.h*.

Its role is to provide a link between the praxis and the SMURPH station (node) that will be running it. This file is only used by VUE^2 (the PicOS compiler never looks at it). Its contents are pretty much praxis-independent and can be viewed as a standard incantation, except for the parent type for *Node* (*NNode* in the above variant). VUE^2 provides a collection of standard node types equipped with some predefined features. For example, *NNode* stands for a node running the NULL plugin for VNETI (NULL Node). Other parent node types are *TNode* (for TARP Node) and *PicOSNode*, which is the most generic node type acting as the base type for all other nodes.

The two files *included* within the *Node* station specify the node contents (its attributes). The first of them, *attribs.h*, must be provided in the praxis directory. It links the global variables and functions of the praxis to the node, such that they become its attributes instead of being global, as in the PicOS incarnation. The second file, *starter.h*, comes from the VUE^2 library. It introduces a single method, named *appStart*, which is needed to start up the node (corresponding to powering on a physical node).

The three methods of *Node* announced in *node.h* are defined in *node.cc*, which in our case may look as follows:

```
#include "node.h"

void Node::setup (data_no_t *nddata) {
       PicOSNode::setup (nddata);
       NNode::setup ();
       init ();
}
void Node::init () {

#include "attribs_init.h"
       appStart ();
}
__PUBLF (Node, void, reset) () {
       NNode::reset ();
       init ();
}
```

Illustration 5: The ontents of *node.cc*.

Again, the contents of this file are directly applicable “as-is” to many praxes. The setup method of *Node* is called by SMURPH when the node is created, and formally its role is that of a constructor. Its argument represents a standard package of data describing a PicOS node, which will be discussed later. This package is passed to the setup method of *PicOSNode*. This method is called only once during the virtual life of the node. Any praxis-specific initialization should be put into the *init* method, which is also called on every (virtual) reset. The file *attribs_init.h* (which should be present in the praxis directory) will provide the initialization code for those praxis-specific attributes of *Node* that must be initialized on every reset.

The file also defines (somewhat cryptically) the *reset* method for *Node*. The method says that upon reset, the *NNode* component should be reset (that component will make sure to reset any lower level components in the inheritance chain), and then *Node's init* should be called to restart the praxis. The way the reset method is defined is characteristic of those *Node* methods that must be directly accessible from the praxis threads. As this problem is general, and it affects a large portion of the overall thread interface to VUE[2], it calls for a detailed explanation.

4.3 Referencing PicOS functions and variables in VUE[2] models

Suppose that a PicoS thread calls some function, e.g., the *receiver* thread calling *tcv_rnp* in Illustration 1. In the PicOS incarnation of the program, this is a global C function provided by the VNETI module. All (or at least the vast majority of) such functions must be turned into *Node* methods when the praxis is transformed into a VUE[2] model. However, a SMURPH thread cannot reference *Node* methods directly: it must use a remote reference in the form of *S–>tcv_rnp (...)*, where *S* points to the current *Node* at which the thread is run. Thus, if the PicOS functions were directly mapped into *Node* methods, the thread code would have to be modified in a way that would render it unusable to the PicOS compiler.

The trick played here consists in renaming those functions upon their conversion to *Node* methods and using the original names as macros that expand into the proper remote references. For example, the VUE[2] incarnation of *tcv_rnp* (defined as a method of *PicOSNode*) is named *_na_tcv_rnp*. Then, there is a macro defined somewhat like this:

```
#define tcv_rnp(a,b) (((PicOSNode*)TheStation)->_na_tcv_rnp (a,b))
```

which converts the original reference to *tcv_rnp* to the new method. For all standard functions of PicOS modeled by VUE[2] this scheme is already implemented. Unfortunately, any functions defined by the praxis (which cannot be truly global)[3] must be transformed manually for this trick.

Thus, it is recommended that praxis functions be declared using one of these two macros (available in PicOS as well as VUE[2]):

```
__PUBLS(ot,tp,nam)
__PUBLF(ot,tp,nam)
```

The first argument is the node type to which the function belongs. Note that generally a VUE[2] model may have to deal with multiple node types. The second argument is the function type, and the last one is the function name). Under VUE[2], both macros expand in exactly the same way. For example, both:

```
__PUBLF (Node, void, myfun) (int a, word b) {
__PUBLS (Node, void, myfun) (int a, word b) {
```

expand into

```
void Node::_na_myfun (int a, word b) {
```

while under PicOS, the expansions are:

[3]Some functions may actually be safe when left as global. This happens when they do not reference any non-automatic variables.

```
void myfun (int a, word b) {
static void myfun (int a, word b) {
```

respectively. Note that *static* in front of a method declaration in C++ has a completely different meaning than a *static* function declaration in C.

By using the above macros for declaring all functions, the praxis will make itself independent of these issues and become VUE² compliant while retaining its PicOS compatibility. Admittedly, this does require some effort. Note that not all functions need to be subjected to this transformation. A function that does not reference external variables (meaning *Node* attributes in its VUE² incarnation) can be left alone. This in fact applies to both functions in Illustration 1, which can be simply left as global in the model.

To return to Illustration 5, the reason why *reset* is declared in the VUE² compliant manner is that the praxis may want to call it explicitly. Indeed, *reset* is a legitimate operation in PicOS and the method in Illustration 5 plays exactly that role. Thus, when a thread executes *reset ( )*, it will reference that method.

The macros transforming standard PicOS operations into the respective methods (including the requisite remote access operator) are defined in file *stdattr.h* in VUEE/PICOS. Praxis program files must include that file (directly or indirectly) to have transparent access to those operations.

Note that the same problem concerns global variables. While under PicOS every thread can reference such variables directly, they must be accessed remotely as *Node* attributes in the VUE² model. The only difference is that there are virtually no “standard” (predefined) variables in PicOS that a praxis thread would want to access,[4] and the problem is practically confined to the variables declared by the praxis.

Nonetheless, global variables cannot be referenced directly from a VUE² compliant thread, and essentially the same trick must be applied here as in the case of functions. Let us have a look at the following macros:

```
#define _da(a)       _na_ ## a
#define _dac(a,b)    (((a*)TheStation)-> _na_ ## b)
```

The above definitions apply to VUE². Here is how the same macros are defined in PicOS:

```
#define _da(a)       a
#define _dac(a,b)    b
```

The VUE² compliant way to declare a global variable is to use the *_da* macro, e.g.,

```
int _da (sfd);
```

This encapsulation has no effect under PicOS, but under VUE² the declaration will expand into:

```
int _na_sfd;
```

which obfuscates the original name of the variable with a prefix. The idea is that the prefixed declarations will be inserted as part of the *Node* class declaration, and the prefixed variables will become *Node* attributes. To reference them, a (PicOS) process will use their original

[4]One exception is *entropy* (perhaps it should be replaced by a function). Also RF drivers reference some standard attributes like *statid* or *backoff*, but from the viewpoint of VUE², these attributes are not part of the platform-independent praxis code.

names, which become replaced with macros. Formally, the proper way to reference variable *sfd* in the above example is:

```
((Node*)TheStation)->_na_sfd)
```

and this is what *sfd* should expand into. The role of *_dac* is to provide a shortcut for such macro declarations. Thus,

```
#define sfd         _dac (Node, sfd)
```

will expand into

```
((Node*)TheStation)->_na_sfd)
```

The first argument of the macro is useful in situations when the network model consist of nodes of several types.

Owing to the initialization problems, the declaration parts for global variables usually require separate code chunks for PicOS and VUE[2], which renders the two macros (i.e., *_da* and *_dac*) rather useless under PicOS (see below).

Step 4: Create the file *globals.h*, which in our case may look like this:

```
#ifndef __praxis_globals_h__
#define __praxis_globals_h__

#ifdef __SMURPH__

#define THREADNAME(a) a
#include "node.h"
#include "stdattr.h"

#define sfd    _dac (Node, sfd)
#define Count  _dac (Node, Count)
#define show   _dac (Node, show)
#define plen   _dac (Node, plen)

#else  /* PICOS */

#include "sysio.h"
#include "tcvphys.h"
#include "ser.h"
#include "serf.h"
#include "form.h"
#include "phys_dm2200.h"
#include "plug_null.h"

int    sfd;
word   Count;
heapmem {10, 90};

#endif /* SMURPH or PICOS */
#endif
```

Illustration 6: The contents of globals.h.

In a sense, this file is complementary to *threadshdr.h* because it accounts for the global objects, i.e., ones to be visible by all threads. Again, it has two parts depending on whether it is being compiled under PicOS or VUE[2]. First note the trivial definition of *THREADNAME*. As we said earlier, there is no need to “adjust” thread names in this case as our VUE² model consist of a single praxis.

The SMURPH section of *globals.h includes* two files: *node.h* (which we saw before) and *stdattr.h*. The latter comes from the VUE² library (directory VUEE/PICOS) and contains definitions of macros for referencing standard functions and attributes from praxis threads (see Section 4.2). That part is followed by a series of macros that play the same role as *stdattr.h* but for the global variables and methods defined by the praxis. Even though, as we said before, the two functions *show* and *plen* need not be turned into *Node* methods, let us do it anyway (it will do no harm) for the sake of illustration.

Note that the same macro *_dac* is applied to variables and functions. This may occasionally lead to a confusion because, for example, any occurrence of *show* as a token in the praxis code will be replaced by:

```
((Node*)TheStation)->_na_show)
```

not only when the token actually looks like a function call. There is a more foolproof, albeit also more complex, way to define such macros, i.e.,

```
#define show(a,b) (((Node*)TheStation)->_na_show (a,b))
```

(see macro *tcv_rnp* on page 12), which, as long as there are no conflicts, is an overkill.

The PicOS section of *globals.h* specifies the same *include* files as the original praxis code. It also explicitly declares those global variables from the original code that were not classified as process attributes in *threadhdrs.h*. Also, if there are any other items required by the PicOS incarnations of the praxis, but useless for the VUE² model, that part of *globals.h* provides a placeholder for them.[5]

Note that *globals.h* does not declare the global variables (or *Node* attributes) for the VUE² model – it only makes them accessible to the threads. It does so, however, for PicOS. At first sight, it would seem natural to put all global declarations into a separate file, e.g.,

```
int    _da (sfd);
word   _da (Count);
```

and use the same file to describe *Node* attributes for the VUE² model as well as the truly global declarations for PicOS. Unfortunately, we cannot really do it this way because of static initializations. The truly global declarations in PicOS may be statically initialized, e.g.,

```
int    sfd = -1;
word   Count = 0;
```

and even if they are not explicitly initialized, they still are initialized implicitly to zero. C++ does not allow us to insert variable declarations like this directly into class declarations. Moreover, there is no implicit initialization (to zero) of not explicitly initialized attributes. This means that there is no way to avoid a separation of the two parts: completely different statements are required in the two cases. Additionally, any truly required initialization (even the implicit initialization to zero) must be handled explicitly in the VUE² case.

Step 5: Create two files: *attribs.h* and *attribs_init.h* to take care of the C++ declarations for global variables (page 11 shows where these files are inserted). They will only be read by the VUE² compiler. The first file may look like this:

[5]In particular, the present version of VUE² implements a single *malloc* memory pool (which is sufficient for all our present MSP430 praxes). Consequently, the *heapmem* declaration is not understood by VUE² (yet).

```
#ifndef __praxis_attribs_h__
       int    _da (sfd);
       word   _da (Count);
       void   _da (show) (word, address);
       word   _da (plen) (char*);
#endif
```

It will be directly inserted into the *Node* class. Note that the two global functions *show* and *plen* become *Node* methods. The second file contains the trivial code to initialize those attributes that must be initialized:

```
_da (Count) = 0;
```

As no initial value is assumed by the praxis for *sfd*, there is no need to initialize that attribute (but we have to know that).

Step 7: Perform some cosmetics with *app.c*. First, rename the file to *app.cc* to make it available to the VUE^2 compiler. This suffix will be OK for PicOS: the compiler will first copy the file to *app.c*, compile it from there, and then discard the copy. Then remove from *app.cc* all *includes* and replace them with these two lines:

```
#include "globals.h"
#include "threadhdrs.h"
```

Note that for PicOS, *globals.h* now provides the previous headers. Also, remove from *app.cc* everything that has been put into the PicOS section of *globals.h*, i.e., the global declarations including *heapmem*.

Modify the headers of all those functions that become *Node* methods under VUE^2. Use the macros introduced on page 12. Thus,

```
void show (word st, address pkt) { ...
word plen (char *str) { ...
```

become

```
__PUBLF (Node, void, show) (word st, address pkt) { ...
__PUBLF (Node, word, plen) (char *str) { ...
```

Remove the declarations of symbolic states (they have been moved to *threadhdrs.h* – see page 9). Finally, add this statement:

```
praxis_starter (Node);
```

as the last line of *app.cc*, following the closing *endthread* of the root process. This is an empty statement under PicOS, and under VUE^2, it expands into:

```
void Node::appStart () { create THREADNAME (root); }
```

providing the *appStart* method (page 11) which is called upon *reset* to start the praxis model.

Step 8: Create one more file, named *root.cc*, to provide the SMURPH program with a Root process. Do not confuse it with the praxis root thread. That file may look like this:

```
#include "node.h"
#include "board.cc"
```

```
process Root : BoardRoot {
        void buildNode (const char *tp, data_no_t *nddata) {
                create Node (nddata);
        };
};
```

Illustration 7: The standard contents of root.cc.

This is the place where the execution of the entire model will commence. The actual code of the Root process comes from the VUE[2] library (type *BoardRoot*) and the present declaration only specifies one (virtual) method needed to instantiate a single node of the model. The first argument of *buildNode* is used to identify the node type (through a textual name) in those scenarios when the model consists of several different node types. This argument is ignored in our case. The second argument points to a standard data structure containing the node parameters extracted from the data file. There is no need to interpret these parameters: they are simply passed to *Node's setup* method (page 11), which in turn will convey them to *PicOSNode* constructor.

To see how the conversion has worked, move to PICOS/Apps/VUEE/ILLUSTRATION and execute *mkmk VERSA2* followed by *make*. This will compile a PicOS version of the praxis creating two image files *Image* and *Image.a43*. Then execute *vuee* in the same directory. This will create an executable VUE[2] model of the praxis in the file named *side*.[6]

5 Models involving multiple praxes

See PICOS/Apps/VUEE/TNP, which is a combination of two separate PicOS praxes called Tags and Pegs. In a nutshell, the idea is to have two separate sets of the files described in Section 4. The names of those files are tagged with suffixes *_tag* and *_peg*, depending to which praxis they belong. The separation is in fact perfect (such that except for renaming the files, the conversion can be carried out exactly as for a single-praxis model) up to the rather trivial place where the two praxes are linked together to constitute a single SMURPH program. This part consists of two files: *common.h* and *root.cc*.

```
#ifndef         __tnp_common_h__
#define         __tnp_common_h__

#include "sysio.h"
#include "board.h"

void buildTagNode (data_no_t*);
void buildPegNode (data_no_t*);

#endif
```

Illustration 8: The connecting header file for Tags and Pegs.

The header file announces two functions that are to be called to set up a node from a given praxis. In particular, *buildTagNode* is the Tag's equivalent of Root's method *buildNode* from page 17. This time, to keep the praxes separated (and to minimize the amount of shared code), we prefer to have a transparent function (rather than a method) for each praxis, such that the Root process need not be aware of the layouts (types) of their nodes. The trivial contents of such a function (see *node_tag.cc*) are of course:

```
void buildTagNode (data_no_t* nddata) {
/*
 * The purpose of this is to isolate the two applications, such that
```

[6]It is *side.exe* under Cygwin.

```
 * their specific "type" files don't have to be used together in any
 * module.
 */
      create NodeTag (nddata);
}
```

which is essentially what we saw in the single praxis case (Illustration 7). The *root.cc* file looks like this:

```
#include "common.h"
#include "board.cc"

process Root : BoardRoot {
      void buildNode (const char *tp,  data_no_t *nddata) {
            if (strcmp (tp, "tag") == 0)
                  buildTagNode (nddata);
            else if (strcmp (tp, "peg") == 0)
                  buildPegNode (nddata);
            else
                  excptn ("Root: illegal node type: %s", tp);
      };
};
```

Illustration 9: The SMURPH root program for Tags and Pegs.

The file provides a definition for the same virtual method of *BoardRoot* as the the single-praxis file in Illustration 7. This time we take advantage of the type argument to decide which node type is being created. As we shall see, *boardRoot* is called internally (by the Root process) as many times as many nodes belong to the modeled network. Node specifications in the data file can use textual identifiers to differentiate among multiple node types.

The Tags and Pegs (TNP) model is considerably more complicated than the simple ILLUSTRATION discussed in Section 4, as it implements two complete and fully functional praxes involving TARP and VNETI. Each of them consists of multiple source files. Note that the source code of TARP (PicOS/Libs/LibComms) has been rendered VUE^2 compliant, such that it is effectively shared between PicOS and VUE^2.

6 Input data

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

```
<network nodes="2">
    <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>400m</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>9600 12 0</frame>
       </channel>
       <nodes>
          <defaults>
              <memory>1124 bytes</memory>
              <power>
                 transmitter power   10.0dBm
                 receiver boost             0.0dB
              </power>
              <preamble>32 bits</preamble>
              <lbt>
                 delay               8msec
                 threshold           -109.0dBm
              </lbt>
              <backoff>
                 min                 8msec
                 max                 303msec
              </backoff>
              <uart rate="9600" bsize="12">
                 <input source="socket"></input>
                 <output target="socket" type="held"></output>
              </uart>
          </defaults>
          <node number="0">
              <location>1.0 4.0</location>
          </node>
          <node number="1">
              <location>1.0 10.0</location>
          </node>
       </nodes>
   </network>
```

Illustration 10: A sample data file.

Each data file describes a single <network>. The mandatory attribute of the <network> element specifies the total number of nodes (of all types). This number remains 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 include 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>.

6.1 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.

6.2 Channel

This element describes a radio channel. It is mandatory, which means that a VUE[2] model requires a radio channel (and at least two nodes). 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 a bit as 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 18, 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 number interpreted as the so-called cut-off distance in meters. The cut-off distance is the minimum separation between a pair of nodes beyond which their mutual interference can be ignored. This value is used to restrict 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 ratio 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 10). If it less than the last value, then the bit error rate is 1, i.e., no reception is possible.

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 18) 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 have the property that flipping a single bit renders the symbol invalid.

6.3 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> item requires one argument, which is the node number. The node numbers are internal SMURPH identifiers of the *stations* implementing the nodes. The 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* argument of <network> (page 18). Each node number from 0 to *nodes*–1 must occur exactly once among <nodes>, not necessarily in any particular order.

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.

```
<power> ... xmt power ... rcv boost ... </power>
```

The element provides two floating point numbers: the default transmitter power and the receiver sensitivity (or receiver boost). The interpretation of these numbers is up to the channel model. Generally (and in particular for the shadowing channel model) the transmitter power is in dBm and the receiver boost is in dB. The received signal is multiplied by the boost at the receiver before its assessment, thus, the boost of zero dB can be viewed as "normal".

```
<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).

All the above sub-elements are mandatory in the sense that each of them must appear either among the elements of <node> or, if absent there, its <default> version will be assumed. A situation when neither <node> nor <default> provide the parameter is treated as an error.

One special mandatory element is:

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

which must be present in every <node> and makes no sense (is ignored) in <default>. It assigns a 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.

The following <node> sub-elements are optional: <uart>, <pins>, <leds>. They augment the nodes with optional components (like UARTs), which can be legitimately absent. These elements are discussed below, each in a separate section.

6.4 The UART

Each node may be optionally equipped with a UART, whose functionality, as perceived by the praxis, accurately mimics the functionality of a standard UART accessible with the *io* operation of PicOS. 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,[7] 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.”[8] 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”>
```

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

[8]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.

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.

6.5 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, terminated with the newline character, in the following format:

```
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.

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 a similar effect. In that case, the <output> specification is ignored if present.

6.6 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" pulse=”1,2” 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 elements that can contribute to such a declaration. 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 requires two pins: the counter and the notifier, the numbers of those pins must be specified with the optional *pulse* attribute. They can be any pins, including ones capable of ADC/DAC operation. Then, 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.[9] 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.

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

[9]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.

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.” Also, there is no “string” source option for <input>. 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 hexadecimal followed in turn by 0 or 1, e.g.,

```
P c 0
P 0 1
P 1e 1
```

A *D* command assigns a voltage to the specified pin. The letter is followed by the pin number in hexadecimal and the discretized voltage as a hexadecimal number from 0 to 0x7FFF corresponding linearly to 0 ... 3.3V.[10]

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. Such a message consists of the following six components:

1. The time in seconds (with millisecond granularity).
2. The total number of pins in hexadecimal.
3. The number of ADC-capable pins in hexadecimal.
4. The number of the affected pin in hexadecimal.
5. The status of this pin (a single decimal digit).
6. The pin value in hexadecimal.

Here we have a few examples:

```
24.556 c 8 4 1 0
3600.120 c 8 7 3 0176
2365.667 c 8 2 0 0
```

[10]No negative voltage is implemented at present (but probably will be later).

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 0x7FFF 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 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 initial series of “updates” for all pins reflecting their current status and values.

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) 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 6), 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 five 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
MOVER	to change the locations of nodes

There van be multiple active instances of any connection type at any given moment. For example, several UARTs belonging to different nodes can be interfaced to agents simultaneously. The above list is likely to grow and new functions to the existing connection types are likely to implemented. The present interface should be viewed as preliminary.

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 1.

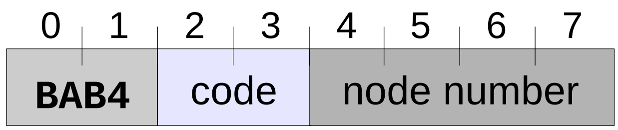

Illustration 11: Connection polling sequence

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 sanity by the VUE^2 program. 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 6.3).

Having received a polling sequence, the VUE^2 program responds with a single byte. If its value is 129 (decimal), it means that the request has been accepted. The remainder of the exchange depends on the request type. If the value is different, it means an error. Having sent the error byte, the VUE^2 program immediately closes its end of the connection. Here is the list of error values:

0	wrong magic sequence
1	node number out of range
2	illegal (unimplemented) request code
3	the node has no UART (for a UART connection request)
4	some agent is already connected to this particular module
6	timeout; this is only sent if a complete 8-byte polling sequence does not arrive within 30 seconds from the moment of connection
7	the module (UART, PINS, LEDS) is present at the node, but it has no socket interface
8	the node has no LEDS module (for a LEDS connection request)
9	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
11	this and the next code are sent within a session, to indicate a protocol error; code 11 means that the request sequence is too long
12	illegal request (sent within a session in response to an illegal request/query)

Code 10 is intentionally not used as it could be mistaken for a newline character, which terminates messages sent by the VUE^2 program in course of normal sessions. A protocol error detected by the program in the middle of a session causes immediate disconnection (EOF) preceded by a single byte (11 or 12).

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 byte 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 6.4). 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 6.4.

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 byte, the VUE^2 program immediately (without polling by the agent) sends the full list of updates reflecting the current status of all pins.[11] Each such a message takes a complete ASCII line of text, as described in Section 6.6, 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 6.6. 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*-type messages, 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 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 confirmation byte, the VUE^2 program sends to the agent the initial configuration of LEDs as a message described in Section 6.5. 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.

7.5 MOVER protocol (request code 4)

This type of connection has no corresponding module in the network and requires no node number (which is ignored, but must be present in the polling sequence). By following this protocol, the agent is able to modify the locations of nodes. Following the confirmation byte sent by the VUE^2 program to the agent, the rest of the conversation is in ASCII, with lines terminated by (single) newline characters.

[11] This may change. It seems that a more concise format of the initial status message is needed.

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 simple command that consists of a single nonnegative number in hexadecimal, e.g.,

```
4E
```

In response, the program will send the following line:

`nn total x y name`

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.

There are two ways for the agent to change a node's position. If a line sent to the VUE^2 program consists of three numbers, e.g.,

```
c3 3.01 6.11
```

where the first number is a nonnegative integer in hexadecimal and the remaining two are nonnegative floating point numbers, it is interpreted as a request to “teleport” the node indicated by the first number to the coordinates (x,y) described by the remaining two. The node immediately disappears at its previous position and shows up at the new one.

Another format of the move request is like this:

`nn x y v g`

where *nn, x* and *y* are as before, i.e., *nn* is the node number in hexadecimal and *x, y* are its new coordinates. This time, however, the move is not instantaneous. The remaining two (floating point) numbers indicate the movement speed (*v*) in meters per second and the granularity of a single step (*g*) in meters. For illustration, suppose that node number 42 (2A in hexadecimal) is located at coordinates 1.0, 4.0. With this request:

```
2A 4.0 4.0 0.5 0.1
```

the node will travel the distance of 3 meters from its old location to the new one at 0.5 m/s, which means that the trip will take 6 seconds. It will consist of a series of teleportations displacing the node by 10 cm at a time, 30 steps altogether, along a straight line.

The minimum possible speed is 10^{-4} m/s and the finest granularity is 10^{-4} m. A smaller number is tacitly replaced by the limit.

Having started a node's trip, the program is ready to accept a new request. This means that a number of trips can be in progress at any given time. They continue towards their target location even after the agent has disconnected. If a node is currently in the middle of its trip and another move request is received for that node (including a teleportation request), the trip is interrupted and the new request is interpreted from where the node has been stopped.

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.

7.6 CLOCK protocol (request code 5)

With this simple protocol, the agent can receive time information from the VUE^2 program. Following the confirmation byte, the program is 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.

8 UDAEMON

Here we briefly describe the functionality of udaemon, which is a Tk program implementing a rudimentary GUI agent for talking to 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).

When started, the program opens a window that looks as shown in Illustration 12. This window is the launcher of five different interfaces, corresponding to the five protocols described in Section 7.

Illustration 12: The root window of udaemon.

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 13. 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).

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 6.4. 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).

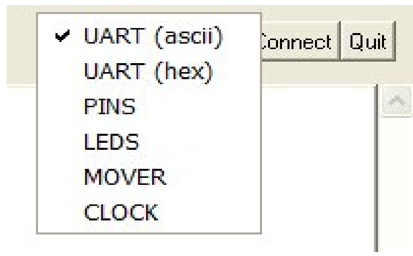

Illustration 13: The interface menu.

A UART connection produces a window shown in Illustration 14. 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 25). The mode switch only affects the data to be displayed, not that already present on the screen.

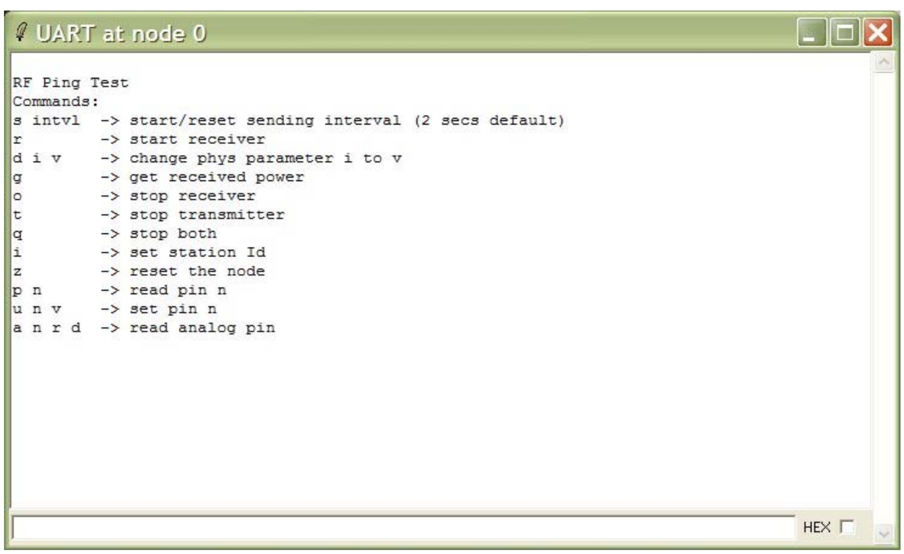

Illustration 14: A UART window.

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.

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 15 pops up on the screen.

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 15, 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 27)

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.

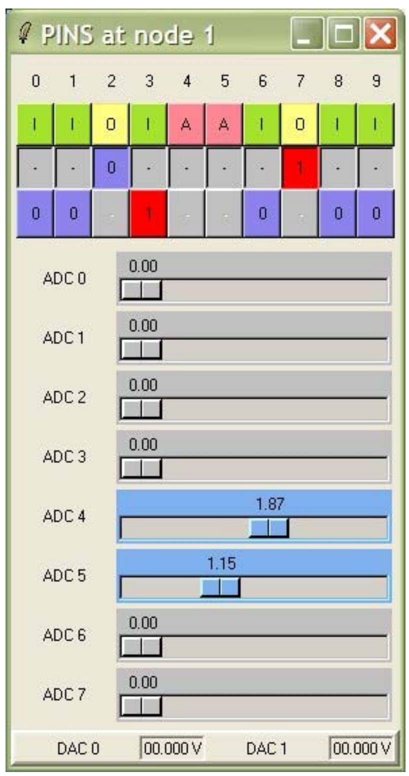

Illustration 15: A PINS window.

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 15). 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 15.

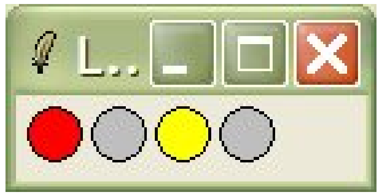

Illustration 16: A LEDS window.

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 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.

8.4 The MOVER interface

This interface takes no node number. Illustration 17 shows a sample window displayed in response to a MOVER 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* does not take advantage of the five-argument move command (see page 32), and each move request is handled as a series of teleportations, whose granularity and range directly follow the manual displacement of the node on the screen.

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 when you release the mouse button. 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.

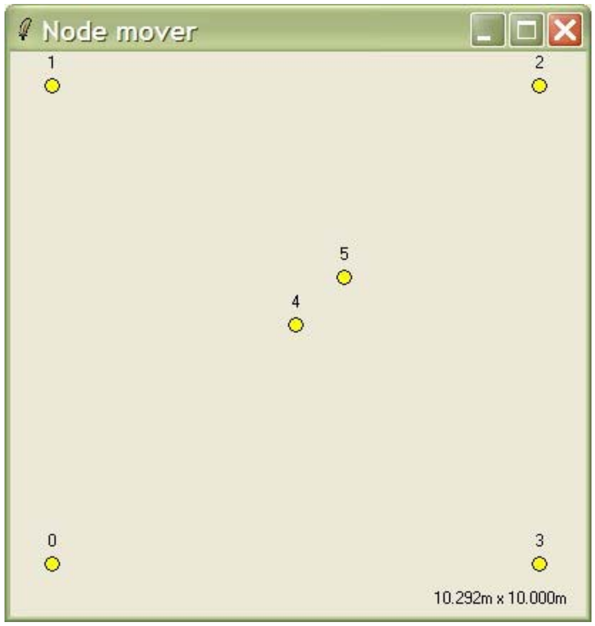

Illustration 17: A MOVER window.

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.

8.5 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.