PiComp

version 0.2

June 2010

Introduction

PiComp performs two types of jobs:

1. Preprocessing source files in "enhanced" PicOS language (see below), such that they can be handled by the (standard) C compiler (say, *gcc* in *mspgcc*).

2. Transforming PicOS praxes (possibly multiprogram praxes) into VUEE models.

Regarding the first type of job, we would like have a more convenient way of expressing certain PicOS constructs than what can be achieved by relying solely on C preprocessor macros. Old versions of some of those constructs, especially process (strand/thread) declarations, look weird. In addition to resolving those problems, a more powerful pre-compiler will allow us to think about new syntactically viable constructs that could be added to the PicOS language to facilitate programming in its true spirit. While, at this time, the contribution of PiComp to enhancing the programming language of PicOS may appear moderate, we may view it as the first step. It may be comforting to know that, should we come up with something more creative, the path to adding new (possibly non-trivial) constructs to the language is essentially open.

For the second role of PiComp, the manual effort needed to maintain VUEE compatibility of PicOS praxes has been quite serious and irritating, despite our attempts to simplify the rules and recommendations. There has been a rather destructive tradeoff between the ease of maintenance (of VUEE-compliant code) and the degree of structural finesse of the program that one was wiling to indulge in. While the most recent set of rules and recommendations for achieving VUEE compliance "manually" may not be excessively complicated, those rules nonetheless force you to organize your program into a rigid collection of files while paying close attention to several counter-intuitive and messy details (like the multiple *contexts* of a variable declaration). The easiest way to avoid mistakes is to minimize the number of all those tricky places that must be carefully checked/updated when modifying the praxis, which in turn stimulates restrictive simplicity, like making all variables global and keeping them in one place.

In this context, the role of PiComp is to transform the source program in PicOS mechanically into a set of files that can be then comfortably (and automatically) processed by the VUEE/SMURPH compiler. The original source program doesn't have to be especially prepared for that task, except for the avoidance of some obscure and basically irrelevant constructs, which reasonable programmers would statistically never use anyway. While I can give you examples where PiComp will fail, there is no need to advertise them formally as something that a programmer should learn and memorize before embarking on a programming task. For one thing, it is hoped that all failures of PiComp to process a correct program will be detected and diagnosed (as opposed to producing incorrect but formally runnable code). Then, the compiler can be extended, if the discovered limitation proves serious enough.

Quick start

Make sure that all three components: PICOS, SIDE, and VUEE have been installed, e.g., according to the rules described in *Installation.pdf*. You need at least CVS release version **R100609A** of SIDE and VUEE (and use the most recent release of PICOS).

Go to subdirectory *Scripts* of *PICOS* and copy two files from there: *mkmk* and *picomp* to your personal *bin* directory (which appears in *PATH*). Remember to use the newest version of *mkmk*.

The VUEE compiler (created by SMURPH/SIDE) in your *bin* directory must be named *vuee*, *vue2*, or *vue*.

Move to *PICOS/Apps/VUEE/TEASER* for a simple example of a multiprogram praxis.

Execute:

```
mkmk WARSAW
```

and things should compile for PicOS (producing the usual Image files).

Now execute:

```
picomp
```

which should produce the *side* executable (*side.exe* under Cygwin). You will also see a directory named *VUEE_TMP*. PiComp places in it the intermediate sources that is submits to the SMURPH/SIDE compiler (*vuee*).

A more elaborate example is in *Apps/VUEE/new_il_demo*. This is Wlodek's IL praxis, which I (rather crudely) converted back to a basic and straightforward (non-VUEE compliant) form using the new PicOS keywords (see below) and added a few comments here and there.

New syntax for some PicOS constructs

Don't panic! The old syntax still works. In particular, if you execute *mkmk* on an old program, followed by *make*, it is going to do the same kind of job it did before. Similarly, when you execute *vuee* on your old, handcrafted, VUEE-compliant praxis, it will work exactly as before.

The newest version of *mkmk* (followed by *make*) will additionally handle programs using the new constructs (described below), but only for the source files with suffix *.cc* (not *.c*). Even for those files, it will do nothing if the new constructs are not there (so they are optional).

However, if you want to use PiComp to do the VUEE conversion for you, the program MUST use the new constructs (and the names of its source files must end with *.cc*).

Another thing: don't try to run *picomp* on a manually converted VUEE-compliant praxis. It won't work. The primary reason is the extra files (which don't really belong to PicOS, but to SMURPH/SIDE). While *mkmk* would walk around them, *picomp* will try to interpret them as bona-fide members of the praxis. IMO, there's no sense to insist on PiComp compatibility with the manual conversion mess.

Here is a sample thread from *PICOS/Apps/VUEE/TEASER*:

```
fsm receiver {
  shared address rpkt;
  entry RC_TRY:
      rpkt = tcv_rnp (RC_TRY, sfd);
  entry RC_SHOW:
      show (RC_SHOW, rpkt);
      tcv_endp (rpkt);
      proceed RC_TRY;
}
```


First of all, the declaration now involves a standard set of braces "`{ ... }`" which are properly matched (no ugly keywords at the end). All threads in PicOS are now declared with the same keyword *fsm*. Strands are differentiated from threads by an argument following the FSM's name, like in:

```
fsm sender (char) {
  address spkt;
  entry SN_SEND:
      spkt = tcv_wnp (SN_SEND, sfd, plen (data));
      spkt [0] = 0;
      strcpy ((char*)(spkt + 1), data);
      tcv_endp (spkt);
      finish;
}
```

The argument must be a type name. As before, it is turned (invisibly) into a pointer type and used to declare a local *data* variable pointing to the process's private data.

Here is how to create a process (a thread in this case);

```
runfsm receiver;
```

and here's how to create a strand:

```
runfsm sender (ibuf + 1);
```

The operation returns a value (the process ID), so it can be used in expressions, e.g.,

```
PId = runfsm receiver;
...
if (runfsm sender (ibuf + 1) == 0) {
...
```

exactly as the previous operations *fork*, *runthread*, *runstrand*.

Note the new syntax of states. You can write:

```
entry name :
```

or

```
state name :
```

with exactly the same meaning. DO NOT *#define* the symbolic constants for the state names! They are taken care of automatically. The first state on the list (the topmost one) is by default the initial state (receives number zero).

Within the FSM, the state names can be used as integer constants (and, for example, assigned to variables or passed as arguments to functions). However, those constants are invisible outside the FSM. Consequently, different FSM's can use the same state names (for different states) without conflicts.

You can precede a state by *initial*, if you want to explicitly mark it as the starting state of the FSM (normally, the lexically first state has this property), e.g.,

```
fsm mbeacon {
    state MB_SEND:
      oss_master_in (MB_SEND, 0);
    initial state MB_START:
      delay (25 * 1024 + rnd() % 10240, MB_SEND);
}
```

In this case, **MB_START** will be the initial state. Only one state can be declared as *initial* in a given FSM.

Note the local variable declaration in *receiver*:

```
shared address rpkt;
```

The keyword *shared* says that the variable is going to survive state transitions while being "shared" by all instances of the process, so, strictly speaking it isn't local, but rather global (although visible only by the instances of this particular FSM). From the viewpoint of PicOS, *shared* is essentially equivalent to *static*. From the viewpoint of VUEE, such a variable is still implemented as a "global" node attribute, except that its name is mangled on a per-FSM basis, which effectively means that different FSM's can use the same name without a conflict.

A digression: I was tempted to implement true local variables, e.g., declared with the keyword *local* or maybe *context*. As a matter of fact, I am still tempted to do it and I probably will. This, however, will require us to revise the concept of local data (which is currently represented by the *data* attribute of a strand). The proper solution should implicitly organize all *local* variables declared by an FSM into a structure that will be pointed to (internally and transparently) by the *data* pointer. That structure will be set upon the FSM's creation and deallocated when the FSM terminates. Any arguments of *runfsm* will be assigned to the local variables in the order of their declaration. This, however, will break the downward compatibility (the internal operation of process creation/termination in the kernel will have to change), and this is the reason why I have been reluctant to include this feature in the first version of PiComp without discussing it (at least with Wlodek). Otherwise, it will be rather simple to implement.

The syntax of *proceed* allows now for this:

```
proceed state_name;
```

in addition to the traditional:

```
proceed (state_name);
```

There isn't much more - just two somewhat exotic keywords relevant to VUEE only. Normally, when you have a *const* variable, and you switch to a VUEE model, you want to have it "constant" on a per node basis (such that different nodes have different views of that constant). This basically means that you want it appear as a node attribute. Consequently, PiComp will transform global *const* variables into node attributes (similar to other, non-*const* variables). In some situations, a *const* is really constant once for all, and corresponds to something that should be hardwired into the code flash of all nodes looking exactly the same everywhere (e.g., a fixed message string). In such a case, you can declare the constant as *trueconst*, as in this example:

```
static trueconst char ee_str [] = OPRE_APP_MENU_A
      "EE from %lu to %lu size %u\r\n";
```


While PicOS will just strip the "true" part of the keyword, VUEE will make sure to treat the declared item (or items) as actual constants (and place them in code rather than at the nodes). Note that *trueconst*, while being formally ignored by PicOS, may play an advisory role (as a useful comment) by referring to those items that are in fact supposed to be identical and immutable across all nodes ever running the program. For example, the program may use constants that are settable on a per-node basis, like a HOST ID being "burned" into the flash but with different values for different nodes. Such (mutable) constants should be declared as *const*, while true constants should be marked as *trueconst*.

Another VUEE-specific (and probably unfortunate) keyword is *idiosyncratic*, which can appear in front of a function definition or announcement, as in:

```
idiosyncratic int tr_offset (headerType*);
```

or

```
idiosyncratic int tr_offset (headerType *h) {
      return 0;
}
```

It is ignored by PicOS, but for VUEE it means that the function should be turned into a node method, rather than being defined as global or static. The difference is mostly irrelevant, with the notable exception of a few TARP functions which are to be provided by the praxis program. The VUEE models of those functions are implemented as *virtual* methods of the node. Thus, their program-specific instances must be also provided as node methods in order to properly substitute for the dummy originals.

I came up with the name *idiosyncratic*, because this prefix will also allow different functions with the same name, appearing in different programs of the same praxis (destined for different node types), to coexist within the VUEE model without upsetting the linker. In this sense, the keyword makes the function idiosyncratic to one program of the (multiprogram) praxis.

One option is to make all functions *idiosyncratic* and forget about the whole issue. I will investigate if there are any problems with that.

Compilation

At the first level of approximation, you may assume that there is nothing special about a PicOS program being prepared for VUEE. As long as all the PicOS components referenced by the program have their VUEE models, it should be pretty much automatic. Needless to say, if you insist on referencing some arcane features (symbols or constructs) associated with a specific board, sensor, etc., you may have to apply conditions, like:

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

which work as before. However, you do not need them for structuring the program into separate blocks for PicOS and VUEE, nor you need to provide any VUEE-specific kludges, glues, headers, etc., as you did with the manual conversion.

Version 0.1 of PiComp (which nobody saw) couldn't handle compound variable initializers, e.g., ones like this:

```
struct MTag t = { 1, 2, 3 };
```

which had to be circumvented with ugly conditional compilation. The present version DOES HANDLE such initializers. The problem with them is that when a variable becomes a node attribute (which happens to all non-automatic variables in a VUEE model), its initialization cannot be carried out statically, so one has to transform such declarative initializers into dynamically executed code. While there is no problem with a simple initializer, e.g.,

```
int sid = -1;
```

which can be trivially converted to an assignment statement, a compound initializer cannot be treated this way. PiComp handles a compound initializer by declaring a dummy *static const* version of the variable (which can be initialized statically) and copying its memory area (*memcpy*) to the actual variable at initialization.

Note that, similar to plain C, you don't have to (dynamically) initialize those components of structures and arrays that are supposed to be zero. VUEE will first zero out the entire set of node attributes (the node's "global" RAM) and then do the detailed initialization.

One general restriction that immediately comes to my mind is that you cannot declare types (*structs*, *unions*, *typedefs*) with the same name in different files of the same program (you still can in different programs of the same praxis). The reason is that wherever you declare your variables in the source program, they are eventually put together as attributes of the same class in the VUEE model. Consequently, all the types used in the declarations of those variables are visible at the same time, and they must not interfere.

PiComp may occasionally complain about something it cannot handle. Here is one example of a PiComp-generated error diagnostic:

```
app_peg.cc:27: picomp prohibits empty size in array typedefs
```

caused by a statement like this:

```
typedef arr_type [][3];
```

This illustrates a restriction on what kind of static/automatic data can be transformed into node attributes with definite memory layouts. Also, this kind of declaration:

```
int my_arr [][3];
```

will yield:

```
app_peg.cc:27: empty size only allowed with an initializer
```

As you can infer from the diagnostic, this:

```
int my_arr [][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

will work fine (PiComp will derive the size along the first dimension of *my_arr* from the initializer).

If you stumble upon something weird, please let me know and I will see what I can do. Note that despite the rather significant transformations undergone by the source program before it is submitted to the VUEE (SMURPH/SIDE) compiler, and subsequently to g++, PiComp tries very hard to tag the generated code with proper line sequencing marks, such that the errors reported by g++ should reasonably closely refer to the original source (which does not mean that they are never confusing).

In terms of its layout as a collection of files, a praxis to be compiled by PiComp should essentially conform to the expectations of *mkmk* (see *mkmk.pdf*). The praxis may consist of multiple programs rooted at files named *app_xxx.cc*, where *xxx* is a program-specific tag (we shall call them *app* files). All files in the current directory (where the compiler has been called) whose names fit this pattern are assumed to represent separate programs of the praxis. if there is only one program, the app file can be named *app.cc*. Any remaining files contributing to those programs, including sources and/or headers, may appear within the same directory (at the same level as the app files) or in any of its subdirectories, except for those named *CVS*, *KTMP*, *ATTIC*, or *VUEE_TMP* (ignoring the case). Headers are sought automatically as needed. A source file must be referenced either from the app file or from one of the other files already deemed to belong to the set (be it a header or a source) via the special comment sequence //+++ (see *mkmk.pdf*). For example:

```
//+++ app_diag_peg.cc "lib_app_peg.cc" <msg_io_peg.cc>
```

adds three source files to the program's set (the delimiters are optional). All the subdirectories of the current directory (with the exceptions noted above) are scanned automatically for files requested this way (as well as for any *#included* headers). There is no need to specify exact paths. For example, the file *lib_app_peg.cc* referenced above can be provided in any subdirectory (at any level) of the praxis's main directory.

When you invoke the compiler as:

```
picomp
```

it will do the following things:

1. Check for the completeness of the environment (in particular VUEE and SIDE).

2. Scan the current directory for app files and build the list of tags representing the programs belonging to the praxis. If there is an app file named *app.cc* (empty tag, a single-program praxis), then no other app files are legal (the compiler will complain if it sees any).

3. For each program, the compiler will parse all source files belonging to the program and create their compiled versions in directory VUEE_TMP. Any headers referenced by those files will be expanded in (so they will not show up in VUEE_TMP). The compiler will add two extra files for each program: the VUEE header (included by all compiled files) and the VUEE initializer (containing code for starting and resetting the node model),

4. Having processed all programs, the compiler will add one more file to VUEE_TMP containing the root process of the SIDE simulator and providing a glue for the (semi-independent) models of different nodes (for the multiple programs).

5. Finally, if there were no errors thus far, the compiler will invoke the VUEE compiler (*vuee*) on the produced set of files. If the compilation succeeds, the executable (*side* or *side.exe*) will be moved to the praxis main directory.

You can say:

```
picomp -n
```

to omit the last step of the above sequence. Note that you can then move (*cd*) to VUEE_TMP and execute *vuee* by hand.

If you do:

```
picomp -- ...
```

then any arguments following `--` will be passed to *vuee* (rather than being interpreted by *picomp*).

When invoked this way:

```
picomp -p
```

the compiler acts as a filter (stdin to stdout) to translate a PicOS source file using the new syntax (*fsm* and so on) into "traditional" PicOS (gcc + macros) code. You are not supposed to ever have to do that by hand. Such calls are inserted by *mkmk* into *Makefiles* at the pertinent stages. If a source file doesn't reference the new keywords, it will be passed unchanged by the filter, which makes the procedure downward compatible without any extra hints. Also note that new and old constructs can be mixed this way (but not if you are compiling into VUEE!)

A bit about internals

I am only discussing here PicOS-to-VUEE translation. The PicOS-to-C filter is much simpler. It uses only one nontrivial component of the PicOS-to-VUEE compiler, which is the declaration parser. Its purpose is to recognize *shared* declarations within an FSM and move them around.

This is all very preliminary. I may write more later. Things may still change without a warning.

The compilation is heuristic, but it should work for all (or most of) sane cases. I am almost sure that the compiler will never produce incorrect code without a warning. Other than that, it may fail on good and healthy source. Whenever that happens, I will do my best to fix the problem, if it can be fixed at all.

This is how it works in a nutshell:

Having determined the list of programs to compile (note that one program may consist of multiple files), PiComp proceeds to handle them one at a time. No information survives from one program to another (note that "program" is not synonymous here with "file"), except for the global list of tags (the *xxx* parts from *app_xxx.cc*), which are needed at the end to generate the root file of the SMURPH simulator (also called the glue).

When processing a program, the compiler starts from its app file. The file is read and scanned for preprocessor directives (conditions and includes). All files *#included* by it are read as needed and processed recursively. An included file that belongs to VUEE

(comes from VUEE/PICOS), as opposed to the praxis, is not fully expanded (it will be included later again for the "true" compilation), but its definitions are preserved (and conditions are obeyed), such that the resultant output file from that stage is devoid of macro references and conditional code. Note that macros must be pre-expanded (before parsing) as they may produce (translate into) PicOS keywords or type declarations.

The stage produces a set of preprocessed (expanded) files comprising the program. Note that all those files are source files (all praxis-specific headers have been already incorporated into them). They may still reference things (but, notably, not macros) that are not defined within them, but all such things share one property: they already belong to the virtual world (i.e., to the model), so PiComp doesn't have to worry about transforming them into anything different from what they are.

Each of those files is subsequently parsed and processed. The processing order is exactly the same as that of their addition to the set (as requested with the special comments //+++), with the app file going first. This is what happens:

All type declarations (*typedef*, *struct*, *union*), including those implicit in variable declarations, are identified and removed. They are stored in a list (eliminating duplicates and detecting any conflicts) to be included as a single set of type declarations for the entire program (to be provided in a special header file). The reason is that the (necessarily single and centralized) declaration of the node class, including all its attributes, must in principle see all those data types.

Function announcements are left intact except for those announcing *idiosyncratic* functions, which are turned into macros transforming references to them into references to the corresponding node methods. Similarly, actual definitions of *idiosyncratic* functions are transformed into mangled definitions of node methods. The signatures of those methods are stored for inclusion (declaration) within the node class.

Global variable declarations are analyzed and split into lists of variables, including their types and initializers. Those lists are stored for inclusion (declaration) as node attributes. The original declarations are removed and replaced with macros referencing those (mangled) attributes via the global node pointer. Static declarations are treated the same way, except that their mangling is file-specific. This way, the same static name appearing in two different files will translate into two different attributes. This closely resembles what you were doing manually, except that now, with all this being done by a program, we can afford to be more exacting and careful.

FSM's are transformed into code methods of SMURPH processes. Information regarding their names, types (thread/strand), and lists of states is stored until all files have been handled. Also, *shared* declarations within FSM's are transformed into macros referencing the appropriately mangled node attributes. Those macros are *#defined* in front of the code method and *#undef(ined)* at its end. This way, the *shared* variables are only visible within the FSM. Note that there is no need to do anything about usual (automatic) declarations. FSM states are collected, to be inserted into the definition of the corresponding SMURPH process class.

PicOS-specific keywords, like *runfsm*, *proceed*, are identified and processed as required.

When all files of the program have been processed, PiComp generates a header file to be included by all of them. That file combines all the declarations of types extracted from all the program files and follows them by the declaration of the node type along with all the attributes and methods. Then it declares the process types for all the FSM's.

Finally, one more source file is created which contains the reset code, i.e., to initialize the node attributes and to start the root FSM. Any dummy variables needed by compound initializers are declared as *static const* within that file.

Having handled all the programs, PiComp creates the glue file containing the root process of the simulator. Then, if there were no errors (and the compiler wasn't called with -n), PiComp invokes *vuee* to compile the contents of *VUEE_TMP*.