Transforming Sources to Petri Nets : A Way to Analyze Execution of Parallel Programs

Jean-Baptiste Voron
Université Pierre & Marie Curie
UMR CNRS 7606, LIP6 / MoVe
4, place Jussieu, Paris, F-75005 France
jean-baptiste.voron@lip6.fr

Fabrice Kordon
Université Pierre & Marie Curie
UMR CNRS 7606, LIP6 / MoVe
4, place Jussieu, Paris, F-75005 France
fabrice.kordon@lip6.fr

ABSTRACT

Model checking is a suitable formal technique to analyze parallel programs' execution in an industrial context because automated tools can be designed and operated with very limited knowledge of the underlying techniques. However, the specification must be given using dedicated notations that are not always familiar to engineers (so far, model checking on UML raises complex problems that will not be solved immediately).

This paper proposes an approach to perform transformation of source code (C programs) into Petri nets, a suitable specification for model checking. To overcome the complexity of the resulting specification, we focus on specific aspects of the program. So, several transformations can be performed to verify some aspects of the processed programs. Parts of this approach could be reused by intrusion detection systems.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.4 [**Software Engineering**]: Software / Program Verification—*Formal Methods, Model Checking*

General Terms

Verification, Reliability, Design

Keywords

Petri Nets, GCC, Software Analysis

1. INTRODUCTION

Behavioral analysis of concurrent systems cannot be completed anymore using only "traditional" test-based approaches. First, their complexity often makes impossible to cover a significant part of the state space by simulation. Second, testing concurrent systems is not trivial and may lead to complex problems like probe effects [24]. To overcome these limitations, it is now recognized for many years that formal methods are of interest since they provide more trustable and mathematically founded information [7, 17].

Among the available formal verification techniques, model checking is particularly interesting due to its potential for full automation as well as its error reporting capabilities [4, 13]. So, neither a long training nor a long practice are required from engineers, using model checkers to extract execution paths leading to undesired behaviors.

However, the problem is about the way engineers design specifications too. Most model checkers require formal specifications as inputs : Automata [2], Promela [21], Petri-nets [15], etc. These input formalisms may be difficult to learn. They usually also propose a low level of abstraction not adapted to their use in the context of industrial-size projects without extensive practice.

A way to avoid the learning of such languages is to consider verification at source program level. This cannot replace a modeling/verification phase but it is a way to tackle the specification problem that is crucial especially in domains related to security [10].

The purpose of this paper is to propose a translation of programs sources into Colored Petri nets for analysis purpose. Such an analysis can be performed to evaluate the behavior of these programs like we proposed in [23] for intrusion detection systems.

To tackle the combinatory explosion of generated models as well as their analysis, we propose to consider separately various *perspectives* of the analyzed program. So, a property will be checked according to the corresponding perspective.

This paper is structured as follows. Section 2 sketches a brief state of the art and presents our objectives. Section 3 explains how we extract the information produced by GCC. Exploitation of this information to produce and optimize Petri nets is detailed in sections 4 and 5. Finally, we apply our technique to an example in section 6.

2. OBJECTIVES

Several methodologies and techniques have been investigated for decades to produce correct software. The objective is to track bugs and imperfections, through program analysis.

International Workshop on Petri Nets Tools and APplications PNTAP 2008, March 3, 2008, Marseille, France.

Common approaches can be broadly divided into two families: (i) testing and (ii) verifying. While *verifying* methods use abstractions of the program, *testing* methods use executable version of the program.

Widely used in software engineering, static code analysis [3] is an example of the second family of program analysis. It consists in looking into the source code for patterns known to generate errors or bad behaviors during execution (such as bad pointer declaration, increment modification inside a loop, etc. . .). Generally associated to compilers, dedicated analyzers handle programs written using most of common languages. Well known examples deal with C/C++ [5] or Java [14, 8] programs. However, since these tools generally focus on syntactic aspects (more than semantic ones) of the program, only few properties can be formalized.

To handle this lack of precision, [12] defines *formal software analysis* and presents model checking [6] as a foundation technique. Instead of considering only the source code, this kind of analysis consists in systematically searching all possible behaviors of a system. It requires a model built from the program to be analyzed and translated into a language that can be processed by a model checker.

Producing program's abstractions for a model checker require skills in model building but also a deep understanding of the program. A bad interpretation of the source code leads to a less accurate model that has bad impact on the verification.

Model checking techniques are used by many teams. Each one usually uses a dedicated modeling language. JavaPathFinder [19] or Brandera [9] translates Java source code directly into Promela language which is the input language of SPIN [21]. Feaver [22] produces Promela from C programs; but the model construction relies on user-specifications that describe pairs of C and Promela patterns. Another approach, SLAM [1], deduces predicate abstraction from a C program ; these skeletons (they only contain boolean instructions) are then used as input to a dedicated model checker.

Software checking is a variation of the previous technique. It is a systematic test at the implementation level. State-space exploration is performed by controlling and observing the execution of all visible operations in the concurrent processes of the system. VeriSoft [16] uses this kind of verification approach.

In addition to difficulties encountered during model construction, the size of the resulting model can also be a problem. For some programs (generally multi-threaded or parallel), we must consider the *state space explosion* problem. Common solution adopted by the community is to decrease the precision of the model, and thus, to reduce the precision of checked properties. Of course this trade-off is not satisfactory when dealing with system security.

2.1 Our objectives

To cope with these challenges, we aim at producing a framework dedicated to program analysis. This framework must be as precise and automatic as possible. Moreover, since we want to perform formal verification we have to choose an intermediate language that can be processed by a model checker.

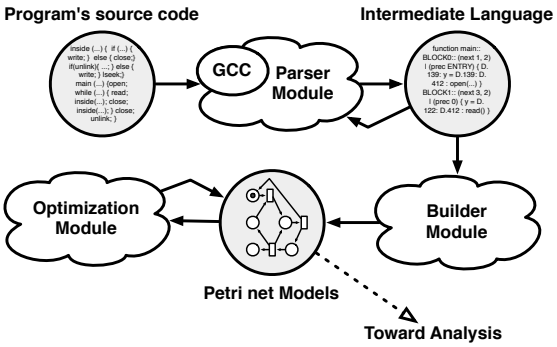


Figure 1: Production factory

Given the size and the complexity of programs we want to transform, we have selected Petri nets as the intermediate representation of the program. First, this formalism captures complex behaviors in a compact way. Moreover, Colored Petri nets is particularly adapted to handle parallel or multi-threaded behaviors (where similar patterns are executed concurrently). Second, the use of Petri nets (more especially *Symmetric nets*[1]) let us benefit from the large collection of provided dedicated tools, like CPN-AMI [25].

2.2 Our approach

To achieve our goal, we design the *production factory*, made of several tools, presented in figure 1. Each one deals with a dedicated transformation.

The process follows three major steps. First, the *parser module* analyzes the program's sources and transforms them into intermediate representations. Then, each representation is interpreted by the *builder module* which produces sets of Petri net models. Finally, to reduce complexity of those models, the *optimization module* executes a set of reductions on each of them (see section 5).

During the process, our production factory considers sources only if they are related to specific behaviors of the studied program. It's up to engineers to decide what behavior is relevant or not.

We call *perspective* each of these specific behaviors and *remarkable element* each related information extracted from the source code. Each perspective has its own set of remarkable elements which could be words, structures or more complex patterns defined into the *perspective definition*.

Considering security related domain, common analyzed perspectives should be: system calls sequences (to avoid race conditions), synchronization mechanisms, array bounds (to avoid buffer-overflows), user-defined invariants (to guaranty security invariants) and i/o behaviors.

[1] *Symmetric nets* were formerly known as *Well-Formed nets*, a subclass of *High-level Petri nets*. The new name was chosen in the context of the ISO standardisation of Petri nets [20].

The final output of our production factory is a set of Petri nets: each one corresponds to a dedicated perspective. Given the analysed program, the set of studied perspectives may change. Engineers may also define their own perspectives. It brings flexibility to our approach and allows finer-grained analysis.

3. ANALYZING SOURCE FILES

Our first concern is to transform the source code into a representation that can be automatically analyzed. Basically, the parser module must reorder the source code and select relevant information.

In our factory, we use an existing compiler framework to do the first part of this process while our component finishes the work.

3.1 Slicing the program using GCC

We choose to design the *parser module* as a wrapper around the GNU Compiler Collection[2] (GCC) in order to analyze source files. This choice gives us some independence from the programming language (C, C++, Java). This operation, called *slicing* [26], is done by one of the several layers of GCC. Among GCC's output, we exploit the control flow graph (CFG) of the program, i.e, all paths that might be traversed through a program during its execution.

More precisely, GCC produces a file describing the program in terms of blocks linked together. These blocks are arranged according to the program's control structures (functions, loops, conditionals...) and are grouped to form function's CFG.

Listing 1 shows some parts of a CFG extracted by GCC. The corresponding C program is presented in section 6.

```
1.   ;; Function beaphilo
2.   # BLOCK 0
3.      # PRED: ENTRY (fallthru)
4.      D.2307 = id + 2;
5.      d = D.2307 % 3;
6.      i = 0;
7.      goto <bb 5> (<L5>);
8.      # SUCC: 5 (fallthru)
9.   # BLOCK 1
10.     # PRED: 5 (true)
11.     if (i != g) goto <L1>; else goto <L3>;
12.     # SUCC: 2 (true) 4 (false)
...
160. ;; Function main
161. # BLOCK 0
162.    # PRED: ENTRY (fallthru)
163.    goto <bb 4> (<L3>);
164.    # SUCC: 4 (fallthru)
...
245. # BLOCK 7
246.   # PRED: 6 (true)
247.   beaphilo (i, &f);
248.   # SUCC: 8 (fallthru)
249. # BLOCK 8
...
257.   D.2689 = fork ();
258.   if (D.2689) goto <L11>
259.   else goto <L10>
...
```

Listing 1: CFG snippet produced by GCC

[2]The parser module uses the `fdump-tree-all` option of GCC which is only available since version 4.0. This version is thus required to make the factory work.

After this step all control structures are rewritten and included in the CFG representation. Consequently, we do not deal anymore with control structures like `for`, `while`, `continue`, `break`... but only with blocks sequences. Even "evil sequences" are simplified by this operation.

3.2 Building a dedicated perspective

As specified in section 2.2, our approach relies on the construction of *perspectives*: only selected ones are processed by the factory to produce Petri nets. One perspective cannot be unselected: the one describing the program's structure (*struct perspective*). Additionally to this unremovable perspective, this paper also considers system calls (*syscall perspective*) and synchronization structures (*sync perspective*).

The *syscall* perspective uses a list of system calls extracted from the system files (available in the `/usr/include` directory). Our bench machine (kernel 2.6.21-2-686) provides a list of 316 system calls. In addition, we look into `libc` to catch common functions using such system calls. This dictionary could be extended if needed.

The *synchronization* perspective uses a dictionary of the structures involved in program synchronization. Moreover, if a program uses a dedicated synchronization library, we can easily upgrade the dictionary to handle a new set of structures.

For each selected perspective, the parser module extracts remarkable elements from the CFG and stores them into separated data files. These files are then transmitted to the builder module.

4. GENERATING PETRI NETS

Once all remarkable elements have been extracted from source code, the *builder module* produces a model for each selected perspective. This model is the result of a merge operation between structure information and considered perspective's information as shown in figure 2.

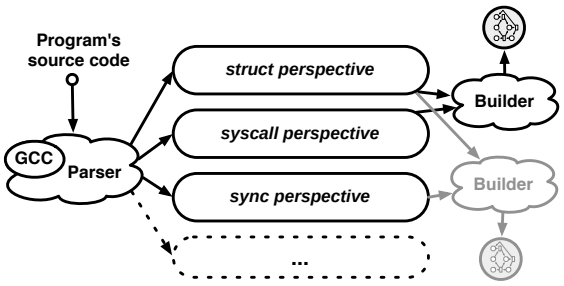


Figure 2: Perspectives increases flexibility

The builder module uses transformation rules to produce models. Each perspective has its own set of rules which are presented and detailled in the following tables.

Table 1 shows the structure of a rule's description. The first row presents the identifier and the title of the rule. The second row is optional and, if present, contains preconditions that must be satisfied before applying the rule. The third row describes the transformation's algorithm itself. It maps

the CFG structure to a Petri net. Finally, the last row puts side by side a CFG snippet and its corresponding Petri net model.

Rule's Identifier: Rule's Title.	
Precond: Rule's preconditions if needed.	
A short transformation description.	
Snippet from a CFG	The resulting Petri net

Table 1: Transformation Rule Definition

Examples in rules are mainly[3] extracted from the CFG snippet presented in listing 1. The parser module has attributed identifiers to functions of the CFG : `beaphilo` function's ID is 0. `main` function's ID is 1.

4.1 Building the structure model

The structure information must be processed first to be reused by other perspective. Its construction relies on seven rules. The first two rules (**Struct1** and **Struct2**) are dedicated to blocks management.

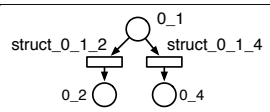
Struct1: CFG blocks.	
We associate a place `F_X` to each block `X` of a function where `F` is the function's identifier calculated by the parser module.	
`;; Function beaphilo` `# BLOCK 0`	

Struct2: CFG functions.	
We create two places for each analysed function `F` in the CFG. They are labeled `F_entry` and `F_exit` where `F` is the identifier attributed by the parser module.	
`;; Function beaphilo`	

In all next rules, and to ease the notation, we will call function `F`, the function whose identifier (attributed by the parser module) is `F`.

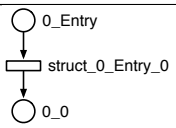
The next two rules (**Struct3.{1,2}**) deal with links between blocks. The first rule browses blocks by following all successors paths. Depending on the configuration, we sometimes need to apply the **Struct3.2** rule to complete the generated model.

Struct3.1: Successor links.	
Given a block `X` of a function `F`. For each block's successor `Y`, we create a transition labelled `struct_F_X_Y`. Finally we link the place associated to the block `X` to the new transition, and the new transition to the place designated by `Y`.	

`;; Function beaphilo`
`# BLOCK 1`
`...`
`# SUCC: 2 (true) 4 (false)`
	

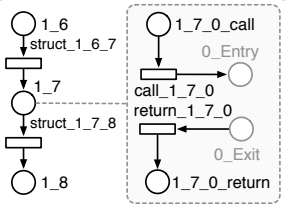
[3]If some elements are missing in the CFG's snippet, explanations which are following the rule complete it.

Struct3.2: Predecessor links.	
Given a block `X` of a function `F`. For each block's predecessor `Y`, we create a transition (if it does not already exist) labelled `struct_F_Y_X`. Finally we link the place associated to `Y` to the new transition, and the new transition to the place designated by `X`.	

`;; Function beaphilo`
`# BLOCK 0`
`# PRED: ENTRY`
	

The next transformation two rules create links between functions. The first one (**Struct4**) generates the link itself while the second controls the return path of a function called from several locations in the program. To do so, we add information to the link structure (**Struct5**).

Struct4: Links between functions.	
Given a block `X` of a function `F` calling a function `G`. We define a sequence of places and transitions associated to the place `F_X` defined as follows: we create two places labeled `F_X_G_call` and `F_X_G_return`. The first one is linked to a transition `call_F_X_G` which is linked to place `G_Entry`. For the return path, the place `G_Exit` is linked to a transition `return_F_X_G` which is linked to the place `F_X_G_return`.	

`;; Function main`
`...`
`# BLOCK 7`
`# PRED: 6 (true)`
`beaphilo (i, &f);`
`# SUCC: 8 (fallthru)`
	

Unfortunately, rule **Struct4** does not fully control return path constructions. If two functions `A` (id=1) and `B` (id=2) make a call to function `C` (id=0), the petri net produced by **Struct4** (figure 3) allows bad return path (in the example, the sequence : `call_1_X_0`, `return_2_X_0` is a mistake).

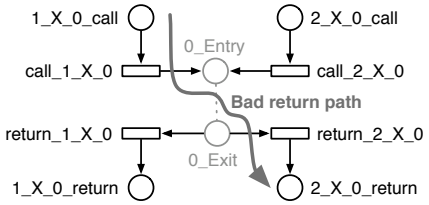


Figure 3: Example of bad return path

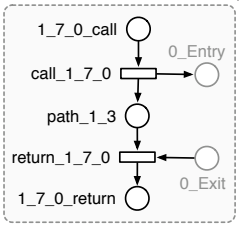
Next rules (**Struct5** and **Struct6**) adress this problem. The following assumptions help us to determine whether the return from a function `C` to a function `A` is valid or not:

- The return is possible if `A` called `C` (**Struct5**)
- The return is possible if all enclosed functions' calls (recursive or not) made in function `C` have returned. (**Struct6**)

Struct5: Avoid bad path return (first assumption).

Each function call from `F` is associated with a place `path_F_C` where `C` is a unique call identifier calculated by the parser module. We create a link from `call_F_X_G` to this new place and from this new place to the `return_F_X_G` transition (see **struct4**).

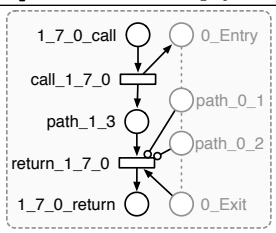
Sequence defined by **Struct4**



Struct6: Handle enclosed calls (second assumption).

For each `return_F_X_G` transition, we create a link from place `path_G_*` to the `return` transition. Instead of a simple arc, this link is an inhibitor arc. Thus the transition is fireable only if all places `path_G_*` are empty.

Sequence defined by **Struct4**



Finally, the initial marking is set up. A simple token is put into the place `X_Entry` where `X` is the identifier of the main function.

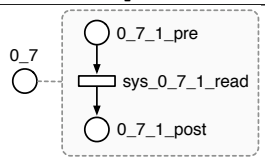
4.2 Building the system call model

As said in section 2.2, a perspective is characterized by a set of remarkable elements to be extracted from source code. In the *syscall perspective*, remarkable elements are system calls.

Syscall0: System call inclusion

Given a system call `S` in a block `X` of a function `F`.
We associate to the place `F_X`, a place `F_X_I_pre`, a transition `sys_F_X_I_S` and a place `F_X_I_post` where `I` is a unique block's instruction identifier. This identifier is given by the parser module. We also create arcs along the path `F_X_I_pre` → `F_X_I_S` → `F_X_I_post`.

```
;; Function beaphilo
...
# BLOCK 7
D.2596 = fd * 8;
read (D.2596, &l, 2);
```



Some types of system calls correspond to specific behaviors. We categorize them to model these specificities. Next rules handle these specific system calls categories. The first one is dedicated to system calls that manage processes.

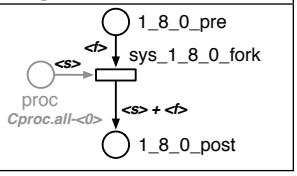
We create a new color class for our models : $C_{proc} = \{0..p\}$ where p is the maximum number of processes the system can handle. This limit can be modified by system designers to fit their specific needs like implicit limitations or special configuration. All places' domain have to be set to $D = C_{proc}$. All arcs carry the process id. We replace in the initial marking the non-colored token by a token from C_{proc} class : $<0>$.

The following rule deals with creation of new processes. It concerns system calls like `fork`, `popen` and `clone`.

Syscall1.1: Process management (*fork behavior*)

Given a `fork` system call numbered `I` in a block `X` of a function `F`. A place `proc` is created (if it does not already exist), initialized to $C_{proc}.all-<0>$[4].
Each time a process is created (i.e. a `fork` is executed), a token is extracted from this place and transmitted to the `post` place. Thus, two tokens are put into the `post` place after the `fork` call: $<f>$ (representing the father process' id) coming from the `pre` place and $<s>$ (representing the son's process id) coming from the `proc` place.

```
;; Function main
...
# BLOCK 8
D.2689 = fork ();
```



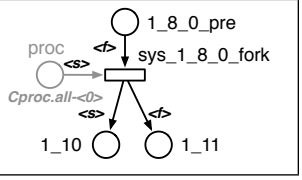
In addition to standard constructions, we have to consider a specific well-known one : `if (fork()) {...}`. Even if we do not yet handle conditional and variables, we must detect this specific case to have a consistent model.

Syscall1.2: *fork* with conditional structure

Precond : `fork` call must be made into a `if` structure

Given a `fork` system call identified by `I` in a block `X` of a function `F`. The `if` structure already provides two output branches. We adapt the model for the `fork` call to this singularity. We create another output arc from the transition `sys_F_X_I_S`. This arc is linked to one of the current block's successors. Its value is $<s>$. The second arc is modified. Its value is now $<f>$.

```
;; Function main
...
# BLOCK 8
D.2689 = fork ();
if (D.2689) goto <L11>
else goto <L10>
# SUCC: 10, 11
```



In case of `clone` system call, the behavior is slightly different because the new process does not execute the code following the `clone` instruction (as the `fork` does) but from the beginning of a designated function `G`.

Syscall1.3: Process management (*clone behavior*)[5]

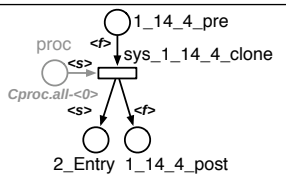
Given a `clone` system call numbered by `I` in a block `X` of a function `F`. The `clone` call designates a function `G` where the new process will begin its execution.

Like the `fork` call, the transition `sys_F_X_I_S` takes a C_{proc} token $<s>$ from place `proc` and a token $<f>$ from

[4]Meaning that `proc` place contains all tokens from C_{proc} class except the $<0>$ one.

place `pre`. When the transition is fired, the token $<f>$ is put into the place `post` and $<s>$ token is put into the place `G_Entry`.

```
;; Function main
...
# BLOCK 14
...
clone(&beaphilo,D.2634)
```

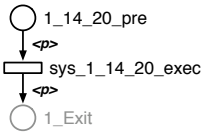


The next category of rules considers system calls that spawn a new program. So far, we do not handle these programs (we could link the model to the one produced for the spawned program but there remain issues to be resolved first). We thus assimilate this type of system call to an `exit`.

Syscall2: Process management (*exec* behavior)[5]

Given an `exec` system call identified by `I` in a block `X` of a function `F`. We delete the place `F_X_I_post` created by the rule **Syscall0**. We create an arc from the transition `sys_F_X_I_exec` to the place `G_Exit` where `G` designates the `main` function.

```
;; Function main
...
# BLOCK 1
...
execv("/bin/ls",D.1301)
```



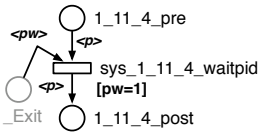
Instructions following the `exec` call are also processed by the builder module because it's not up to it to decide whether an instruction is reachable or not. In the meantime, the arc between the `exec` call and remaining instructions has been removed. Thus, all instructions will be considered as dead code during the analysis.

A third category of rules groups system calls like `wait` or `waitpid` that waits for the termination of a subprocess.

Syscall3: Process management (*wait* behavior)

Given a `wait` system call identified by `I` in a block `X` of a function `F`. We create an arc from the place `G_Exit` to the transition `sys_F_X_I_wait` where `G` designates the `main` function. To be fired, the transition must take one C_{proc} token $<pw>$ from the place `G_Exit`.

```
;; Function main
# BLOCK 11
...
D.2622 = 1;
waitpid (D.2622, 0B, 0);
```



It exists another possible behavior associated with this category of system calls. Indeed, `wait` call blocks the execution until the termination of a process but also until the reception of a signal. This case is handled in the *signals perspective* to avoid confusion in our model structures.

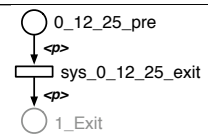
The last category of transformation rules brings together system calls that interrupts the execution of the program such as `exit`.

[5]This rule is not used in our CFG. Example is fictive.

Syscall4: Process management (exit)

Given an `exit` system call identified by `I` in a block `X` of a function `F`. We delete the place `F_X_I_post`. We create an arc from the transition `sys_F_X_I_exit` to the place `G_Exit` where `G` designates the `main` function.

```
;; Function beaphilo
...
# BLOCK 12
...
exit(D.2645)
```



4.3 Building synchronization model

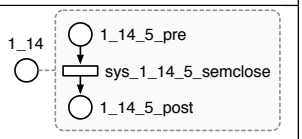
We proceed as for the *syscall* perspective and define several categories of remarkable elements. Each one is associated to a particular Petri net model. We describe, in this paper, POSIX synchronization mechanisms since they are well understandable. System V mechanisms can be handled like them thenceforth the mapping between these two standards have been done.

The first rule inserts the synchronization primitives into the model produced by the *struct* perspective. Some synchronization primitives will then be processed by other rules according to their specificity.

Sync0 : Insertion of synchronization primitives

Given a system call `S` in a block `X` of a function `F`.
We associate to the place `F_X`, a place `F_X_I_pre`, a transition `sys_F_X_I_S` and a place `F_X_I_post` where `I` is a unique block's instruction identifier. We also create arcs along the path `F_X_I_pre` → `sys_F_X_I_S` → `F_X_I_post`.

```
;; Function main
...
# BLOCK 14
...
sem_close(D.3212, ...)
```

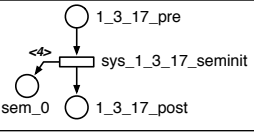


The first category of synchronization concerns semaphores creation (primitives like `sem_init` or `sem_open`). Due to the semaphore initialization, we sometimes need to ask the system designer about an initial value.

Sync1 : Creation of a semaphore

We associate a place `sem_K` to each created semaphore. This place is initialized with the parsed initial value. The valuation v of the input arc of `sem_K` correspond to this one.
When the initial value associated with the semaphore is static, we can easily deduce v (as in the example below). Otherwise, an estimation of the value must be provided by the system designer.

```
;; Function main
...
# BLOCK 3
...
D.3212 = sem_init(...,4)
```

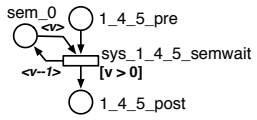


We define a new color class: $C_{sync} = \{0..n\}$. It represents the *capacity* of a semaphore place and should be used by system designers to bound the system.

Once created, program's semaphores are associated with their model thanks to their CFG's identifier. This way, the builder module can identify them when they are used by other synchronization mechanisms in the program.

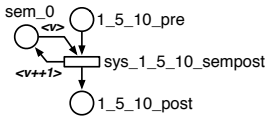
The two next rules deal with semaphore manipulation (primitives `sem_wait` or `sem_post`).

Sync2 : Decrease a semaphore	
The `wait` operation decrements the value of the token in the place `sem_K` representing the semaphore. Thus, the transition `F_X_I_semwait` takes the colored token from `sem_K` place and checks whether its value is greater than 0. The token is then decremented and put back into `sem_F`. Otherwise, the execution cannot be fired until the token value in incremented.	

`;; Function main`
`...`
`# BLOCK 4`
`...`
`sem_wait(D.3212)`
	

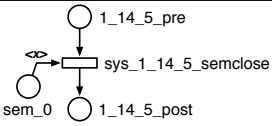
Considering the `try_wait` mechanism (that tries to decrease the semaphore without blocking the execution if it fails), we must decrease the value of the associated semaphore without blocking the execution of the program. This is done by removing the guard $[v > 0]$ from the transition and by changing the valuation of its ouput arc from $<v–1>$ to $<v>$.

Sync3 : Increase a semaphore	
The opposite operation increases the value of the token in the place `sem_K` representing the considered semaphore. This operation is done when the transition `F_X_I_sempost` is fired.	

`;; Function main`
`...`
`# BLOCK 5`
`...`
`sem_post(D.3212)`
	

The last rule deals with semaphore destruction primitives like `sem_destroy`, `sem_close` or `sem_unlink`. Modeling these operations should help designers to detect uses of semaphores while, on the other hand, they have been removed.

Sync4 : Destruction of a semaphore	
The related transition consumes the token in place `Sem_K`. Then, any further access will generate a deadlock that can be detected at verification time.	

`;; Function main`
`...`
`# BLOCK 14`
`...`
`sem_close(D.3212)`
	

4.4 Merging perspectives

The Petri net construction is performed as follows: first, the Petri net model is elaborated from the *struct perspective*. Then, the other perspectives plugged into this first model. The other perspectives are seen as refinements of some places in the structure model.

Refined subnets must respect one constraint: they have only one input place and only one output place. Then these subnets are merged to the structure model as follows: all input arcs to the original place are connected to the only input place in the subnet and all output arcs from the original place are connected to the only output place in the subnet.

Rule **Syscall0** is a typical example of such a refinement: place `0_7` is refined into a two-places/one-transition subnet.

5. REDUCING PETRI NETS

Thanks to all these rules, our factory produces detailed models that are generally quite large (see table 2 in section 6.4). Most places and transitions come from the model of the *struct perspective* since additions from other perspectives are usually rewriting of existing objects. However, many of these elements are not relevant anymore because they do not correspond to remarkable elements we want to observe.

We must remove these useless parts in the specification to reduce its size. Thus, we apply a set of reductions rules on the produced model. The two first rules are slightly adapted from the ones of [18] to fit our strategy. The third one detects a typical configuration that frequently happens in our models. Reductions only concern places and transitions produced by the *struct* perspective.

Pre-agglomeration of transitions This operation aims at reducing sequences of transitions. When a place p is accessed by the firing of a transition t and left by the firing of any output transition of p, we delete the place p and the transition t and make arcs between input places of t and output transitions of p. This reduction is valid only with transitions labeled as `struct_X_Y_Z`.

Post-agglomeration of transitions This optimization is used on conditional structures. When all branches of a transition t join up with the end of a block (which is linked to another block), we directly link all branches to the next block. The intermediate place is deleted.

***Diamonds* reduction** This optimization concerns control structures likes `switch`. When no remarkable elements is located in `case` blocks, a place p_1 is linked to several transitions t_i which are all linked to a place p_2. We call this configuration a *diamond*. Since all paths are equivalent in our model, we merge all transitions into a single one. This way, we have the following sequence: p_1 linked to t linked to p_2.

Our factory uses a dedicated module to perform these optimizations. During the optimization phase, all transitions in the model are checked to detect if reductions can be operated. Once a reduction is applied, impacted parts of the Petri net are reprocessed by the optimization module.

6. APPLICATION TO AN EXAMPLE

We use a simple C program that implements the philosopher problem [11] to illustrate our approach. This example mixes control and processes structures. Moreover, some typical properties can be checked especially those concerning process scheduling.

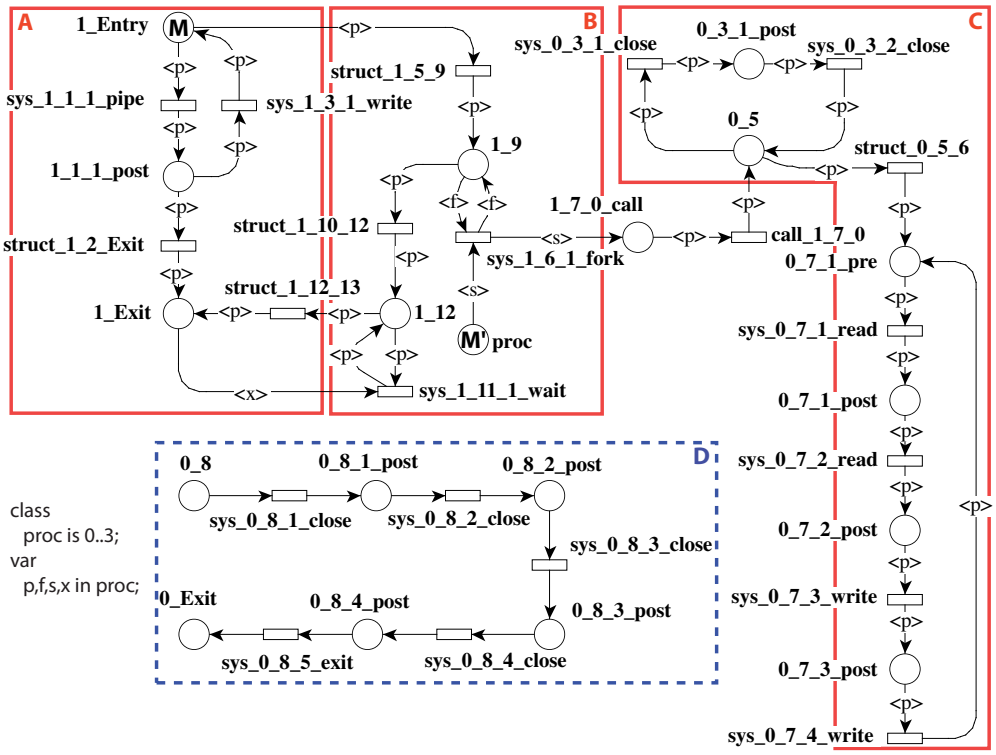


Figure 4: The Petri net produced from the philosopher example

6.1 The program

This program is presented in listing 2. It has been written without any consideration for the modeling part. Basically, the main program starts several processes that will execute the `beaphilo` function implementing one philosopher's behavior.

```
1  #define N 3
2  // Behavior of a philosopher
3  void  beaphilo  (int  id,  int  *f)  {
4   int i,g=id, d=(N+id-1)%N; char l[3];
5   // Part C
6   for (i=0; i<N; i++)
7    if ((i!=g) && (i!=d)) {
8     close((f+(i*2))[0]); close((f+(i*2))[1]);
9    }
10  for (;;)  {
11   if (read((f+(g*2))[0],l,1)>=1) 1;
12   if (read((f+(d*2))[0],l,1)>=1) 1;
13   write((f+(g*2))[1],"o",1);
14   write((f+(d*2))[1],"o",1);
15  }
16  close((f+(g*2))[0]); close((f+(g*2))[1]);
17  close((f+(d*2))[0]); close((f+(d*2))[1]);
18  exit(0);
19 }
20 // Create philosophers as independant processes
21 int main (int argc, char** argv)  {
22  int i, proc[N], f[2*N];
23  // Part A
24  for (i=0; i<N; i++) {
25   if  (pipe(f+(i*2))) return EXIT_FAILURE;
26   write((f+(i*2))[1],"o",1);
27  }
28  // Part B
29  for (i=0; i<N; i++)
30   if (!(proc[i]=fork())) beaphilo(i,f);
31  for (i=0; i<N; i++)  wait(NULL);
32 }
```

Listing 2: A philosopher program

Four parts are identified to help distinguish the mapping between the source code and the generated model (figure 4). Parts A,B and C are highlighted by some comments in the source code. Part C regroups all philosopher's behavioral instructions while part B is responsible for creation of those philosophers. Part A is dedicated to initialization. Part D regroups dead code instructions. In practice, this part of Petri net model is deleted during the building phase.

6.2 The generated Petri net

We apply our production factory to this program and enable *struct* and *syscall* perspectives construction. The result is the Petri net displayed in figure 4.

The place `proc` has no equivalent in the program since it is the result of some transformation rules. It represents a pool of *pids* (deduced from rule **Syscall1.2**).

As planned, the model contains a color class: *proc* that represents the list of available process identifiers. Bounds of this color class are provided by the system designer. In this example *proc*'s bound can be deduced from `N` in the program). Thus, the initial marking is the following:

- **M** = <0> (the main process),
- **M'** = <*proc.all*>\<0>
  (the pool of processes without the main process),

Table 2 shows the model's size after the building phase and after the optimization phase. The model of figure 4 is the reduced one.

	Building	Optimizing	*Reduction rate*
Places	38	13	*65%*
Transitions	44	16	*63%*
Arcs	82	35	*57%*

Table 2: Model's size before and after optimizations

6.3 Some analysis of the model

Some bad constructions such as structural infinite loops or structural dead code can be automatically detected on the reduced model.

Structural infinite loops correspond to a cycle without exit conditions in the Petri net model. Structural dead code corresponds to a subnet that is not connected to the main one. For example, the dotted part in figure 4 corresponds to dead code.

Then the model can be analyzed with Petri net tools. We use CPN-AMI [25] for that purpose. Structural bounds computed on the unfolded net show that only the main process can access to place `1_EXIT` (i.e. the only possible marking for this place is for value <0>). This is the normal behavior we expect since philosophers never stop thinking and eating.

Model checking detects one unique *terminal state* in the system. This situation corresponds to an error during the initialization of structures (Part A in figure 4). In this case, no children are created and the only process (the main one) dies.

We propose to set up an observation point at line 13 (it corresponds to place **0_7_1_post**) to exhibit potential deadlocks. We then associate the following observation formula: *All processes can reach simultaneously this line* (meaning that all philosophers are trying to get their second fork).

This formula is easily translated by a reachability formula to be processed by a model checker. Prod [27] in CPN-AMI returns the following results (with 3 philosophers):

- it is possible to reach this problematic state,
- this sequence of 16 transition firings leads to this state:
  1. **struct_1_5_9**,
  2. **sys_1_6_1_fork**,
  3. **call_1_7_0**,
  4. **struct_0_5_6**,
  5. **sys_0_7_1_read**,
  6. **sys_1_6_1_fork**,
  7. **call_1_7_0**,
  8. **struct_0_5_6**,
  9. **sys_0_7_1_read**,
  10. **sys_1_6_1_fork**,
  11. **call_1_7_0**,
  12. **struct_0_5_6**,
  13. **sys_0_7_1_read**

From such a path, it is possible to locate and to outline and to animate the corresponding instructions in the C program by using the information stored in the CFG.

6.4 Transforming larger programs

Since our global approach is dedicated to intrusion detection system, our benchmarks are more system-oriented than the philosopher problem. Here are some examples of UNIX programs we have processed considering the *struct* and *syscall* perspectives. Results are presented in table 3.

	whois	**ping**	**gzip**
Program's size (lines)	874	2454	7323
Model size (nodes)	1499	2348	5692
Optimized model size (nodes)	627	1037	3301

Table 3: Modeling results for some UNIX programs

7. CONCLUSION

In this paper, we have presented a way to automatically translate a C program into Colored Petri net for analysis purpose. Such an analysis is operated in the context of Intrusion Detection Systems (IDS) where it is of interest to check programs with regards to "dangerous" behaviors. This technique, called *off-line monitoring*, is similar to performing model checking on programs.

We use GCC as a front-end to perform program slicing. We exploit information from the Control Flow Graph (CFG) to produce our Petri nets. So, if experimentation in the paper is done on C programs, our technique should be applicable to any language processed by GCC without majors changes.

To reduce the size of the resulting Petri nets, we consider separate perspectives on a program. A perspective groups remarkable elements to be observed in the target Petri net model. Perspectives can be operated separately or chained, according to what has to be observed.

Our transformation process relies on rules. Rules are associated to a perspective. Our Petri net generator applies the rules associated to the selected perspective. Once the Petri net is generated, we apply an optimization phase that mainly relies on Haddad's reductions [18].

These rules have been implemented in a tool prototype for experimentation. Section 6 shows how our technique can be applied to a simple program, up to some analysis phase. It also shows that bigger programs from the standard Unix library can be processed.

Future work concerns the development of new perspectives such as signals, handling of variables or check of arrays bounds.

8. REFERENCES

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. *SIGPLAN Not.*, 36(5):203–213, 2001.

[2] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification:*

Model-Checking Techniques and Tools, chapter Model Cjhecking, pages 39–46. Springer Verlag, 2001.

[3] D. Binkley. Source code analysis: A road map. In *FOSE '07: 2007 Future of Software Engineering*, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[4] L. Brim. Parallel model-checking. *ERCIM news, special section on Automated Software Engineering*, 58:35, July 2004.

[5] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[6] E. Clarke, O.Grumberg, and A. Peled. *Model Checking*. MIT Press, 2000.

[7] E. Clarke, J. Wing, and et al. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[8] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens. Improving your software using static analysis to find bugs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 673–674, New York, NY, USA, 2006. ACM Press.

[9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[10] H. Debar. An introduction to intrusion-detection systems. In *Proceedings of Connect'2000, Doha, Qatar, April 29th-May 1st, 2000*, 2000.

[11] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[12] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, and W. Visser. Formal software analysis emerging trends in software model checking. In *FOSE '07: 2007 Future of Software Engineering*, pages 120–136, Washington, DC, USA, 2007. IEEE Computer Society.

[13] S. Edelkamp, S. Leue, A. Lluch-Lafuente, and W. Visser. Dagstuhl Seminar on Directed Model Checking, April 2006.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[15] C. Girault and R. Valk. *Petri Nets for Systems Engineering*. Springer Verlag - ISBN: 3-540-41217-4, 2003.

[16] P. Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[17] J. Gogen and Luqi. Formal methods: Promises and problems. *IEEE Software*, 14(1):75–85, 1997.

[18] S. Haddad. A reduction theory for coloured nets. In *Advances in Petri Nets 1989, covers the 9th European Workshop on Applications and Theory in Petri Nets-selected papers*, pages 209–235, London, UK, 1990. Springer-Verlag.

[19] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *STTT*, 2(4):366–381, 2000.

[20] L. Hillah, F. Kordon, L. Petrucci, and N. Trèves. PN standardisation : a survey. In *International Conference on Formal Methods for Networked and Distributed Systems (FORTE'06)*, pages 307–322, Paris, France, September 2006. IFIP.

[21] G. Holzmann. *The SPIN model checker*, chapter An Overview of PROMELA, pages 33–72. Addison-Wesley, 2004.

[22] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 597–607, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[23] F. Kordon, J.-B. Voron, and L. Iftode. Rapid Prototyping of Intrusion Detection Systems. In *Proceedings of the 18th International Workshop on Rapid System Prototyping*, pages 89–96, Porto Alegre, Brazil, 2007. IEEE Computer Society.

[24] H. Krawczyk and B. Wiszniewski. *Analysis and Testing of Distributed Software Applications*. Taylor & Francis, Inc., Bristol, PA, USA, 1998.

[25] Move-Team. The CPN-AMI Home page, `http://www.lip6.fr/cpn-ami`, 2006.

[26] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[27] K. Varpaaniemi. Prod: An advanced tool for efficient reachability analysis. `http://www.tcs.hut.fi/Software/prod`.