

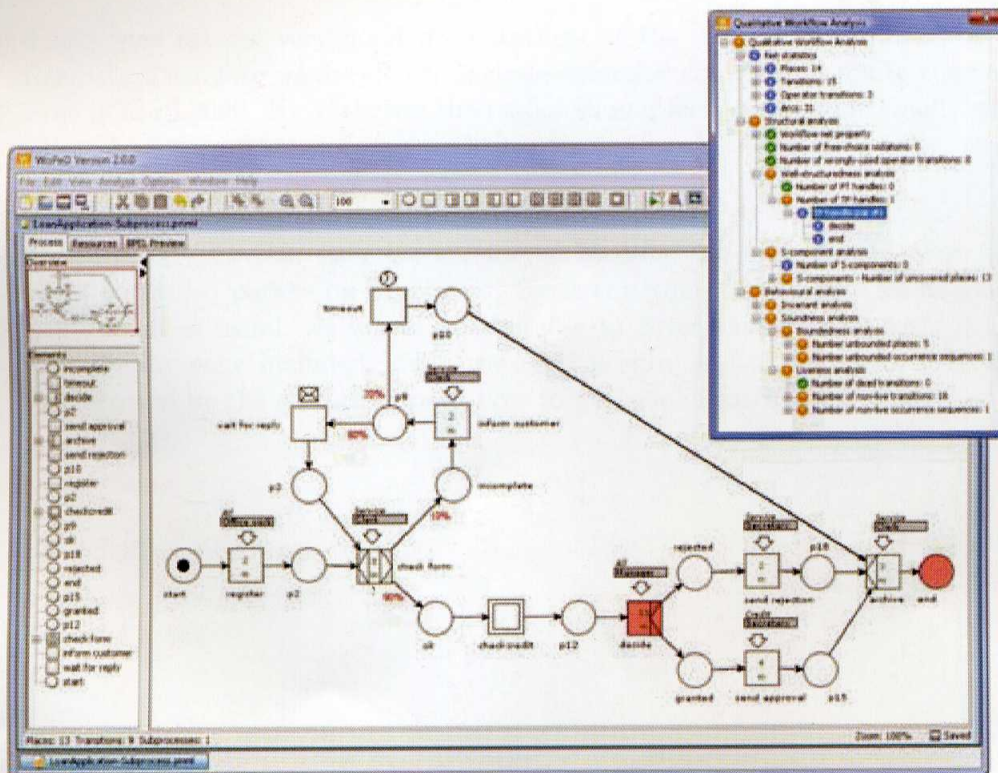
Petri Net Newsletter

Newsletter of the
Special Interest Groups on

Volume 75
ISSN 0931-1084

Petri Nets and
Related System Models

October 2008



Evinrude: A Tool to Automatically Transform Program's Sources into Petri Nets

Jean-Baptiste Voron & Fabrice Kordon

Université Pierre & Marie Curie
UMR CNRS 7606, LIP6 MoVe
4, place Jussieu, Paris, F-75005 France
jean-baptiste.voron@lip6.fr & 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 and its implementation as a tool to perform transformation of C source code 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. Hence, we never model the entire processed program, but only its relevant parts. In this paper, we will apply this approach on some examples using our tool: *Evinrude*.

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 [1]. 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 [2, 3].

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, 5]. 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 [6], Promela [7], Petri nets [8], 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 [9].

The aim of this paper is to propose an automatic 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 [10] 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.

In addition to the theoretical description, this work has also been implemented as a tool named *Evinrude*. It basically takes a C program as input and produces a Petri net ready to be analyzed. The architecture of this tool and some experiments are presented in order to illustrate the approach.

This paper is structured as follows. Section 2 sketches a brief state of the art and presents our objectives. Section 3 presents the *Evinrude* tool. Section 4 explains how we extract the information produced by GCC. Exploitation of this information to produce and optimize Petri nets is detailed in sections 5 and 6. Finally, we apply our technique to an example in section 7.

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.

2.1 Related work

Widely used in software engineering, static code analysis [11] is a technique that looks for patterns known to generate errors or bad behaviors during execution (such as bad pointer declaration, increment modification inside a loop, etc...) into either the source or the object code. Most of languages such as C/C++ [12] or Java [13, 14] are handled by common analyzers which are generally associated to compilers.

In addition to errors finding, static code analysis can also be used in association with formal methods that can mathematically prove properties about a given program. Hence, [15] defines *formal software analysis* and presents model checking [16] as a foundation technique for software engineering. Instead of considering source code only, this kind of analysis consists in systematically searching all possible behaviors of a system.

Model checking techniques are used by many software teams. Each one usually uses a dedicated modeling language. JavaPathFinder [17] or Bandera [18] translates Java source code directly into Promela language, the input language of

SPIN [7]. Feaver [19] produces Promela from C programs; here, model construction relies on user-specifications that describe pairs of C and Promela patterns. Another approach, SLAM [20], deduces predicate abstraction from a C program; these skeletons (they only contain boolean instructions) are then used as input to a dedicated model checker. VeriSoft [21] follows an approach based on a variation of previous model checking techniques. Indeed, it uses systematic testing 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 uses this kind of verification approach.

However, producing program's abstractions for a model checker require skills in model building and 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.

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.

2.2 Objectives of this work

To cope with these challenges, we aim at producing a framework dedicated to (i) program modeling and (ii) bugs and imperfections tracking through model analysis. It should be able to deal with large and/or multi-threaded programs.

For the *modeling part*, our objective is to build models to perform formal verifications. Model construction must be as automatic and precise as possible. Moreover, to handle large programs, we want to provide engineers with a flexible way to select sets of program's specific behaviors they want to analyze.

For the *analysis part*, we focus on the reusability of existing analysis tools and methods. Our factory must build models that can be processed by an existing model checker (as it is done in most of the cited related works) and be used by usual tools dedicated to formal verification.

From all these requirements, we have selected Petri nets as the intermediate representation of the program. First, this formalism captures complex behaviors in a compact way. In particular, colored Petri nets are particularly adapted to handle parallel or multi-threaded behaviors (where similar patterns are executed concurrently). Second, the use of Petri nets (more specifically *Symmetric nets*¹) let us benefit from the large collection of provided dedicated tools, like CPN-AMI [23] or GreatSPN [24].

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 [22].

3 Tool Overview

We have designed a tool to achieve detection of bugs and imperfections in programs: *Evinrude*. It is made of several components as shown in figure 1. Each one deals with a dedicated transformation which is a part of the entire transformation process.

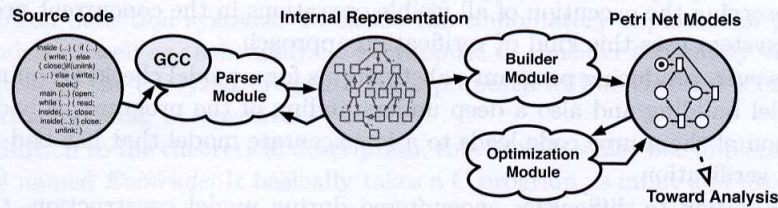


Fig. 1. *Evinrude* tool's overview

We highlight three major steps in this process. First, the *parser module* analyzes the program's sources and transforms them into an internal representation (a kind of rich abstract syntax tree) suitable for other modules (section 4). Then, each of these representations is interpreted by the *builder module* which produces a set of Petri nets (section 5). And finally, to reduce nets' complexity, the *optimization module* applies reductions techniques (section 6).

The exhaustive analysis of an entire program is very difficult, if it is not impossible, since the number of instructions to be checked and properties to be verified is huge and not well bounded. Consequently, we define a *modular* modeling and a *modular* analysis dedicated to specific program's behaviors. Given two different programs, the set of studied behaviors may not be the same: some of behaviors must be studied for both, others not. For example, some behaviors are only relevant for programs that provide networking or parallel features.

Thus, it's up to engineers to decide what are the behaviors they want to study. To help them, we provide (packaged with the tool) a set of common behaviors that should be studied. Engineers can also define their own if it is necessary. This approach brings flexibility to our modeling and allows finer-grained analysis. Considering security related domain, common analyzed behaviors 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.

During the process, our framework considers program's sources and instructions only if they are related to studied behaviors of the program. We call *perspective* each of these specific behaviors and *remarkable element* each related information extracted from the source code for this particular behavior. Each perspective has its own set of remarkable elements which could be words, structures or more complex patterns defined into the *perspective definition*².

² Transformation rules are also defines in perspective definition (see section 5)

The final output of our production factory is a set of colored Petri nets: each one corresponding to a dedicated perspective. Once all nets have been generated, it is possible to merge some of them (or all of them) into a single Petri net to be processed by analysis tools.

4 Analyzing source files

Our first concern is to transform the source code into a representation that can be automatically analyzed by dedicated tools. Considering our *modular* objective, the parser module must select relevant information according to engineers specifications. Since we are not confident enough about compilation process, our factory uses an existing compiler framework to do the first part of this process while our component finishes the work.

4.1 Slicing the program using GCC

The *parser module* is a wrapper around the GNU Compiler Collection³ (GCC) in order to analyze source files. This choice gives us some independence from the programming language (C, C++, Java, etc.) thanks to the various front-ends available in this collection.

The very first operation, called *slicing* [25], is done by one of the many layers of GCC. Among GCC's output, we exploit the *link report*, that defines relations between all sources files, and 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 into CFG functions. Listing 1.1 shows some parts of a CFG extracted by GCC. Functions and blocks are well visible. The corresponding C program is presented in section 7.

```

1.  ;; Function philosopher1
2.  # BLOCK 2
3.  # PRED:ENTRY(fallthru)
4.  printf("&\"Philosopher 1...
5.  goto <bb 4> (<L1>);
6.  # SUCC:4(fallthru)
7.  # BLOCK 3
8.  # PRED: 4 (true)
9.  pthread_mutex_lock(&fork3);
10. pthread_mutex_lock(&fork1);
11. printf("&\"Philosopher 1...
12. pthread_mutex_unlock(&fork3);
13. pthread_mutex_unlock(&fork1);
14. # SUCC:4(fallthru)
15. # BLOCK 4
16. # PRED:2(fallthru) 3(fallthru)
17. D.3892 = food_on_table();
18. f = D.3892;
19. if (f != 0) goto <L0>;

20. else goto <L2>;
21. # SUCC:3(true) 5(false)
22. # BLOCK 5
23. # PRED:4(false)
24. printf("&\"Philosopher 1...
25. pthread_exit (0B);
26. # SUCC:EXIT

26. ;; Function main
27. # BLOCK 2
28. # PRED:ENTRY(fallthru)
29. pthread_mutex_init(&food,0B);
...
36. pthread_create
   (&phils,0B,philosopher1,0B);
37. D.3880 = &phils[1];
...

```

Listing 1.1. CFG produced by GCC

³ The parser module uses the `fdump` option of GCC available since version 4.0.

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

4.2 Building a suitable representation

The builder and the optimization modules cannot directly use the CFG representation (even if it is much simpler than the source code itself). Thus, the parser module has to transform all gathered information into an internal representation suitable for other modules (see figure 2).

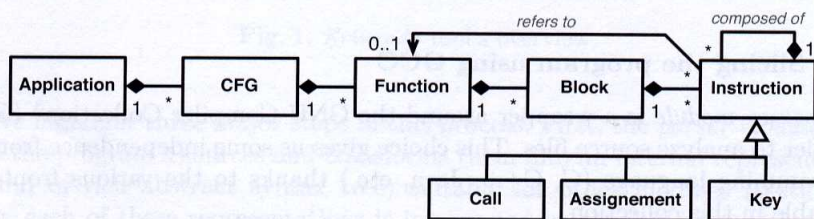


Fig. 2. Internal representation of an analyzed program

Objects `Assignment` and `Call` are more complex in the tool than presented here. However, considering the example and the selected perspectives, this representation is detailed enough to understand the process.

While building this representation the parser module extracts a symbols table and a set of statistics about the program and all its dependencies. It also gives advices on what perspectives should be requested for building and analysis. This list only includes basic perspectives, and engineers can add theirs if necessary. Table 1 presents these results for the studied example program.

Table 1. Information returned by the parser module

Application's name:	dining_philosopher (1 file)
Number of processed CFGs:	1 (320 instructions)
Number of processed functions:	8 (5 connected - 1 main function)
Recommended perspective selection:	- structural perspective (<i>struct</i>) - system calls perspective (<i>syscall</i>) - pthread perspective (<i>thread</i>)

The representation can be saved into a XML file to be processed or re-processed later without having to parse again the entire application source code.

5 Generating Petri Nets

As soon as the internal representation has been produced by the parser module, the *builder module* produces a model for each selected perspective. The *struct perspective* is always processed (and processed first) by the builder module since it deals with the program structure. The result is the skeleton of the final model.

Models, generated using other perspectives, are linked to this skeleton. The structural model and all other perspective's models are then flattened to produce the final Petri net (see figure 3).

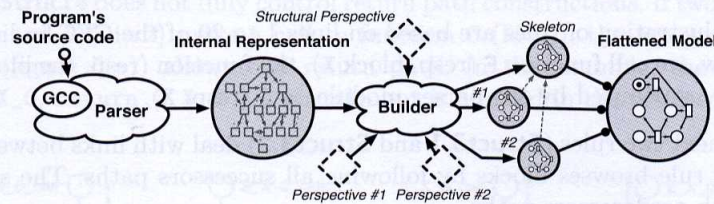


Fig. 3. Perspectives increase the flexibility

In addition to remarkable elements, a perspective definition comes with a set of transformation rules that associate remarkable elements to a set of actions to be applied on the Petri net model (see section 3). The builder module uses these transformation rules to produce all models.

A rule is defined by the following elements:

- its identifier and title
- if necessary, preconditions that must be satisfied prior to application
- the transformation algorithm

Instead of directly manipulating Petri nets, the builder module uses an internal representation using objects (see figure 4). This representation is dedicated to manipulation and optimization of the model.

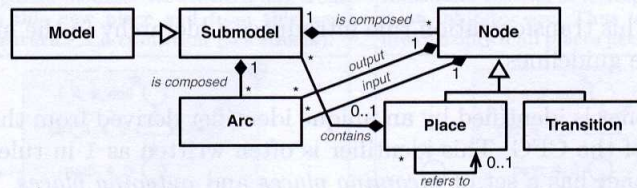

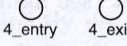


Fig. 4. Description of the internal representation used by the builder module

This representation brings *hierarchy* to Petri nets. Indeed, a model (or a submodel) can contain one or more submodels connected by means of places. A place can also refer to another one to make links between submodels.

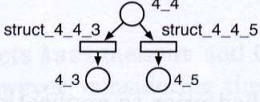
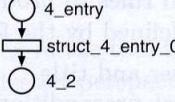
5.1 The structural perspective

The structural (*struct*) perspective contains seven rules. The first two rules (**Struct1** and **Struct2**) are dedicated to blocks management.

Struct1: CFG blocks.	Struct2: CFG functions.
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.	We create two places for each analyzed function F in the CFG. They are labeled $F.entry$ and $F.exit$ where F is an ID attributed by the parser module.
	

Notes: Illustration of rules are based on lines 1 to 20 of the CFG in listing 1.1⁴. From now, we call function F (resp. block X), the function (resp. the block) whose identifier, attributed by the parser module, is F (resp. X).

The next two rules (**Struct3.1** and **Struct3.2**) deal with links between blocks. The first rule browses blocks by following all successors paths. The second one deals with predecessors paths.

Struct3.1: Successor links.	Struct3.2: Predecessor 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 .	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 .
	

Rule **Struct4** creates links between functions. A link between two functions exists only if both have an internal representation available (i.e. their source code is in the program and not in a library). Otherwise, the call is considered as an external call, and might be processed later by another perspective. The Petri net example for this rule is built from line 17 of listing 1.1⁵.

Hierarchy: This transformation rule introduces a hierarchy in the model according to precise guidelines :

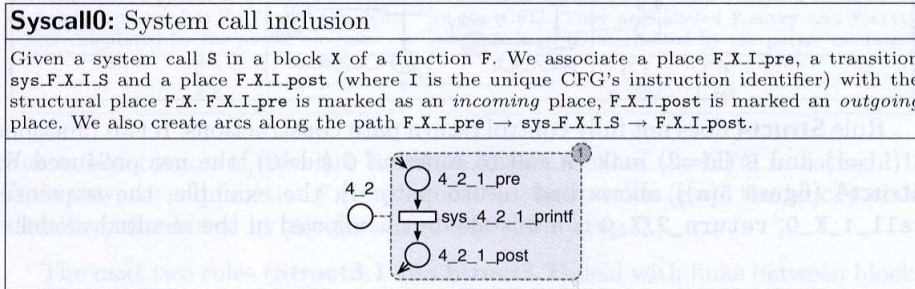
- Each subnet is identified by a unique identifier derived from the instruction counter of the CFG. This identifier is often written as I in rules definition.
- Each subnet has a set of *incoming places* and *outgoing places*. These places are used to merge the subnet into the main net at the end of the build stage.
- A subnet can contain *virtual places* that refer to existing places inside the main net. Virtual places and their references are merged at the end of the build stage too.

⁴ The parser module has attributed id 4 to `philosopher1` function

⁵ The parser module has attributed id 7 to the function `food_on_table()`

5.2 The system call perspective

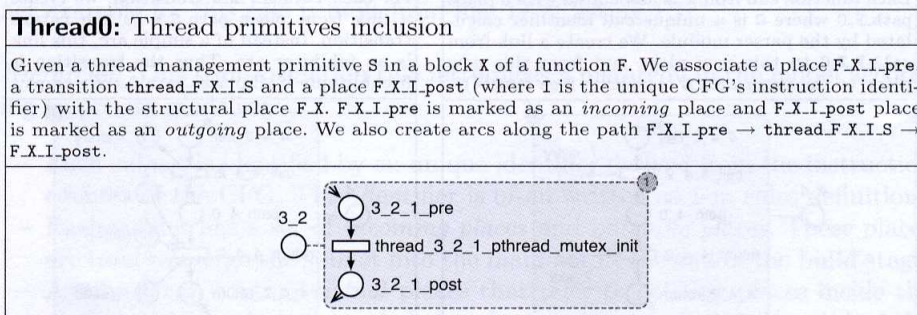
System calls are remarkable elements of the *syscall perspective*. For each system call, a dedicated subnet is created and associated with a place of the structural model (rule **Syscall0**).



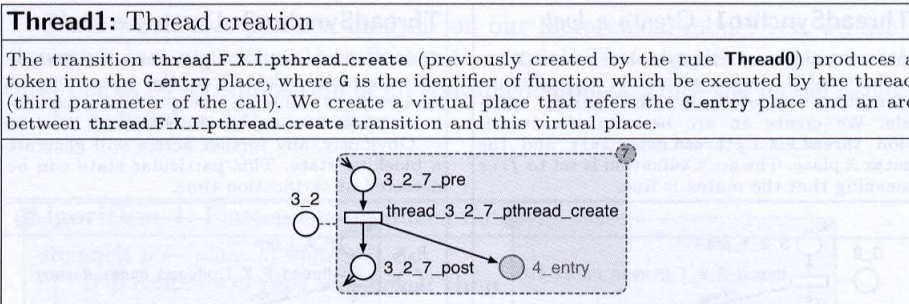
5.3 The thread perspective

Management of threads inside a C program usually involves three primitives: `pthread_create`, `pthread_exit` and `pthread_join`. The *thread-perspective* is designed to handle this kind of behaviors. Other primitives like `pthread_kill` or `pthread_self`, which are also defined in POSIX library, are not handle by this perspective since they do not directly modify the behavior of a thread. Moreover, since it exists three different kinds of thread mutexes, we choose to only analyze the behavior of the most portable one (that is used by default when dealing with the library): `PTHREAD_MUTEX_NORMAL`

Rule **Thread0** handles the creation of dedicated subnets, associated to a structural place. As other perspectives, these subnets are merged at the end of the build phase (see section 5.4).

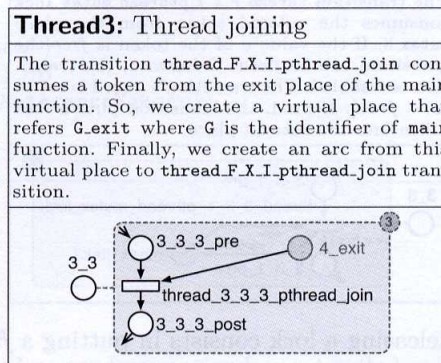
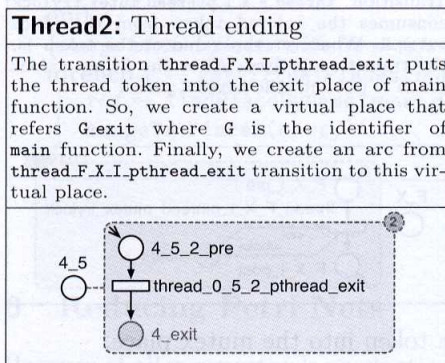


Rule **Thread1** handles thread creation. A new thread is modeled as a new token in the Petri net. Each token represents the state of a particular thread as the program counter does in the system.



Rules **Thread2** and **Thread3** deal with thread termination. In a system, when a thread dies, a part of its information is kept until another thread joined it. After that, dead thread's information is totally removed from the system.

Thus, when a thread ends, its corresponding token is put into the *exit* place of the *main* function, waiting to be retrieved (joined) by another thread (see rule **Thread2**). The *join* operation (which is a system blocking operation) is modeled as trying to get a thread token from the *exit* place of the *main* function (see rule **Thread3**).



Synchronization: Thread synchronization is handled by means of five primitives dedicated to lock management.

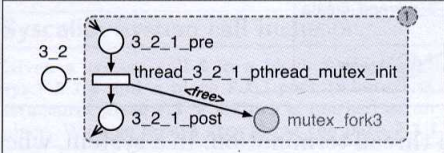
- `pthread_mutex_init()` and `pthread_mutex_destroy()`
- `pthread_mutex_lock()` and `pthread_mutex_trylock()`
- `pthread_mutex_unlock()`

Rules **ThreadSynchro1** and **ThreadSynchro2** focus on mutex creation and destruction. Mutexes are represented by a colored place typed by: $C_{mutex} = \{lock, free\}$. Status of a mutex is represented as follow:

- the mutex is locked when its corresponding place contains token *lock*,
- the mutex is unlocked when its corresponding place contains token *free*,
- the mutex has not been initialized when its corresponding place is empty.

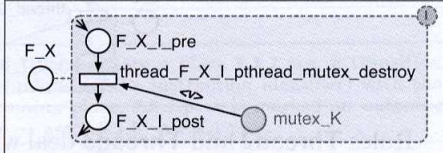
ThreadSynchro1: Create a lock

We associate an empty place `mutex_K` to each created lock, where `K` is a unique identifier computed by the parser module. We create an arc between the transition `thread_F_X_I pthread_mutex_init` and the `mutex_K` place. The arc's valuation is set to `free` meaning that the mutex is free.



ThreadSynchro2: Destroy a lock

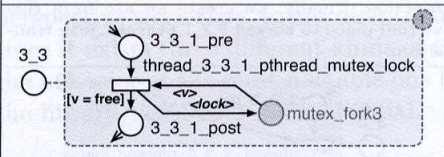
Transition `thread_F_X_I pthread_mutex_destroy` consumes the token from the place `mutex_K` whatever its value is. We create an arc between these two objects. Its valuation is set to `v`. Obviously, any further access will generate a blocking state. This particular state can be detected at verification time.



Locking and unlocking primitives are handled by the next two rules. We distinguish two behaviors when a lock is set. The first one (`lock` primitive) implies a blocking state when the lock is already taken. The second one (`trylock` primitive) is a non-blocking operation. Thus we design two versions of rule **ThreadSynchro3** to handle these two behaviors.

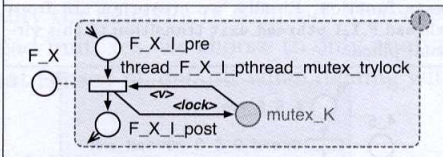
ThreadSynchro3.1: Take a lock

The transition `thread_F_X_I pthread_mutex_lock` consumes the colored token from the place `mutex_K`. If the value `v` of the token is `free` the transition can be fired. Otherwise, the transition is disabled. The transition's guard is set to `[v = free]`. If fired, the transition puts a `lock` token into the `mutex_K` place.



ThreadSynchro3.2: Try to lock

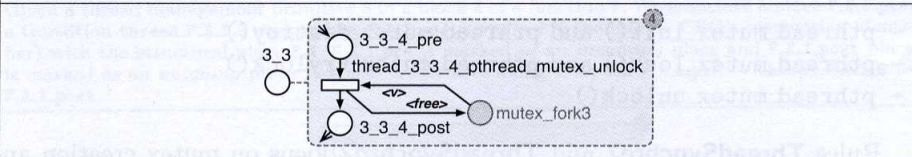
Transition `thread_F_X_I pthread_mutex_trylock` consumes the colored token from the place `mutex_K`. Whatever the value of the token is, the transition is fired. After the firing, a `lock` token is put into the `mutex_K` place.



Releasing a lock consists in putting a `free` token into the mutex place.

ThreadSynchro4: Release a lock

The transition `thread_F_X_I pthread_mutex_unlock` consumes the token of place `mutex_K`. Whatever its value is, the transition is fired and a `free` colored token is put into the place `mutex_K`.



5.4 Merging perspectives

At the end of the building phase, the resulting Petri net is made of:

- a skeleton, thanks to structural perspective;
- several subnets sticked to some structural places.

Since we do not perform analysis on our hierarchical model yet, we use an algorithm (see algorithm 1) to flatten the model. This algorithm replaces structural places by a composition of all contained subnets. Once the model is flat, it can be reduced and then analyzed.

Algorithm 1: Flatten the produced model

```

foreach  $p \leftarrow$  place of main net do
  if  $p$  contains at least one subnet then
     $outgoings \leftarrow p$ 
     $ouputs \leftarrow$   $getOutputTransitions(p)$ 
    foreach  $s \leftarrow$  subnet inside  $p$  do
      { copy all elements from the subnet to the main net }
       $copyAll(s, main)$ 
      { create links between two sets }
       $createLinks(outgoings, getIncomingsPlaces(s))$ 
       $outgoings \leftarrow getOutgoingsPlaces(s)$ 
    end
     $createLinks(outgoings, ouputs)$ 
  end
end
{ merge virtual places with their references }
foreach  $v \leftarrow$   $getVirtualPlaces(main)$  do
   $ref \leftarrow findPlaceByName(main, getReferenceName(v))$ 
   $mergeTwoPlaces(v, ref)$ 
end

```

6 Reducing Petri Nets

Because of all perspective's production rules, our factory produces detailed models that are generally quite large (see table 6). But, while places and transitions that come from perspectives selected by engineers are useful for analysis, objects from the structural model are not relevant anymore; especially because these elements do not correspond to remarkable elements we want to observe.

Thus, we remove these useless parts in the model to reduce its size. To do so, we apply a set of reductions rules on the produced model. The two first rules are slightly adapted from the ones of Haddad [26] to fit our strategy. The third one detects a typical configuration that frequently happens in our models.

Pre-agglomeration of transitions 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 is dedicated to 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 concerns control structures like `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. We then obtain the sequence: p_1 linked to t linked to p_2 .

7 Application to an example

We use a simple C program implementing the well known philosopher problem [27] to illustrate our approach. This code is a variation from the one used by Sun Microsystems to benchmark their Thread analyzer [28].

7.1 The program

The program is presented in listing 1.2. The only change we made to original program is a duplication of the philosopher code (presence of three `philosophers` functions) to avoid data-flow analysis that is not yet processed by *Evinrude*.

```

1  #define FOOD 50
2
3  pthread_mutex_t foodlock;
4  pthread_mutex_t fork1,fork2,fork3;
5  pthread_t p[3];
6  void *philosopher1 ();
7  void *philosopher2 ();
8  void *philosopher3 ();
9  int food_on_table ();
10
11 int main (int argc, char **argv) {
12     int i;
13
14     pthread_mutex_init(&foodlock, NULL);
15
16     // Forks...
17     pthread_mutex_init(&fork1, NULL);
18     pthread_mutex_init(&fork2, NULL);
19     pthread_mutex_init(&fork3, NULL);
20
21     // Philosophers
22     pthread_create(&p[0], NULL, philosopher1, NULL);
23     pthread_create(&p[1], NULL, philosopher2, NULL);
24     pthread_create(&p[2], NULL, philosopher3, NULL);
25
26     pthread_join(phils[0], NULL);
27     pthread_join(phils[1], NULL);
28     pthread_join(phils[2], NULL);
29 }
30
31 void * philosopher1 () {
32     int f;
33     printf("Philo_1_sitting_down_to_dinner.\n");
34     while((f = food_on_table())) {
35         pthread_mutex_lock(&fork3);
36         pthread_mutex_lock(&fork1);
37         printf("Philo_1_eating.\n");
38         pthread_mutex_unlock(&fork3);
39         pthread_mutex_unlock(&fork1);
40     }
41     printf("Philo_1_is_done_eating.\n");
42     pthread_exit(NULL);
43 }
44
45 void * philosopher2 () {
46     int f;
47     printf("Philo_2_sitting_down_to_dinner.\n");
48     while((f = food_on_table())) {
49         pthread_mutex_lock(&fork1);
50         pthread_mutex_lock(&fork2);
51         printf("Philo_2_eating.\n");
52         pthread_mutex_unlock(&fork1);
53         pthread_mutex_unlock(&fork2);
54     }
55     printf("Philo_2_is_done_eating.\n");
56     pthread_exit(NULL);
57 }
58
59 void * philosopher3 () {
60     int f;
61     printf("Philo_3_sitting_down_to_dinner.\n");
62     while((f = food_on_table())) {
63         pthread_mutex_lock(&fork2);
64         pthread_mutex_lock(&fork3);
65         printf("Philo_3_eating.\n");
66         pthread_mutex_unlock(&fork2);
67         pthread_mutex_unlock(&fork3);
68     }
69     printf("Philo_3_is_done_eating.\n");
70     pthread_exit(NULL);
71 }
72
73 int food_on_table () {
74     static int food = FOOD;
75     int myfood;
76
77     pthread_mutex_lock(&foodlock);
78     if (food > 0) { food--; }
79     myfood = food;
80     pthread_mutex_unlock (&foodlock);
81     return myfood;
82 }

```

Listing 1.2. A philosopher program

Basically, the main program starts several threads executing a `philosopherX` function implementing one philosopher's behavior.

We have drawn areas in the Petri net to outline major components they represent according to the original program's function :

- thick plain places represent mutexes,
- zone A corresponds to the philosopher1 function,
- zone B corresponds to the philosopher2 function,
- zone D corresponds to the philosopher3 function,
- zone C corresponds to the food_on_table function,
- other places correspond to the main function.

Table 2 shows the reduction factor of the model during the optimization phase.

Table 2. Model's size before and after optimizations

	Before optimization	After optimization	Reduction factor
Places	99	36	63%
Transitions	91	33	63%
Arcs	226	110	51%

7.3 Some analysis of the model

Some bad constructions such as structural infinite loops or structural dead code can be automatically detected on the model. Infinite loops corresponds to a cycle without any exit condition in the Petri net model. Structural dead code corresponds to a subnet that is not connected to the main one.

Table 3. Details of terminal states found by the model checker

Terminal state #1		Terminal state #2	
Marked place name	Marking	Marked place name	Marking
State_3_exit	•	State_3_2_121_post	•
State_mutex_fork2	<i>free</i>	State_mutex_fork2	<i>lock</i>
State_mutex_fork1	<i>free</i>	State_4_3_151_post	•
State_mutex_fork3	<i>free</i>	State_mutex_fork1	<i>lock</i>
State_mutex_foodlock	<i>free</i>	State_mutex_fork3	<i>lock</i>
		State_5_3_193_post	•
		State_6_3_235_post	•
		State_mutex_foodlock	<i>free</i>

For fine grained analysis, we use the CPN-AMI [23] Petri net tools. Let us generate the state space and check for any terminal state. The state space produced by Prod [29] in CPN-AMI has 845 nodes and 2413 arcs. Two terminal states are detected. They are presented in table 3.

Terminal state #1 corresponds to a normal end of the program since the exit place of the main function (`3.exit`) has only one token (there is no marking in the threads subnets) and all shared resources are free.

Terminal state #2 corresponds to a deadlock. The program's state is the following (we provide instruction identifiers in the CFG when relevant):

- all forks are handled by philosophers,
- philosopher 1 is waiting on instruction #151,
- philosopher 2 is waiting on instruction #193,
- philosopher 3 is waiting on instruction #235,
- main thread is waiting on instruction #121.

Evinrude is able to recognize an instruction identifier and to provide engineers with its translation into C code. This code is extracted from the detailed CFG produced by GCC. Thus, some syntax variations can happen. In the example, table 4 shows associations returned by the tool:

Table 4. Relation between instruction identifier and C code

State ID	Associated C code
121	<code>pthread_join (phils[0], 0B);</code>
151	<code>pthread_mutex_lock (&fork1);</code>
193	<code>pthread_mutex_lock (&fork2);</code>
235	<code>pthread_mutex_lock (&fork3);</code>

The tool also highlights a sequence of 22 transition firings that leads to this deadlock. From such a path, it is possible, as done before, to locate, outline and animate the corresponding instructions in the C program by using the information stored in the detailed CFG. Table 5 produces this trace in an comprehensive way for an engineer.

7.4 Transforming larger programs

Since our global approach is dedicated to intrusion detection systems, our benchmarks are more system-oriented than the philosopher problem and difficult to present in a regular paper. Indeed, attackers often exploit flaws in programs to get control of a computer, stole important data, etc. Generating a model of such programs can lead to detection of flaws or possible entry points for attackers and to prevent these kinds of attacks.

Evinrude is able to deal with large and real programs. Here are some examples of programs we have processed considering only the *struct*, *syscall*, *processes* and *thread* perspectives.

gzip (v1.2.4) is a compression utility included in most of Unix systems. It has been adopted by the GNU projects. It also often uses by FTP servers.

Table 5. A sequence of instructions leading to the deadlock

Main	Philo 1	Philo 2	Philo 3
pthread_mutex_init (&foodlock, 0B); (#112)			
pthread_mutex_init (&fork1, 0B); (#113)			
pthread_mutex_init (&fork2, 0B); (#114)			
pthread_mutex_init (&fork3, 0B); (#115)			
pthread_create (&phils, 0B, philosopher1, 0B); (#116)			
pthread_create (D.3880, 0B, philosopher2, 0B); (#118)			
		D.3901 = food_on_table (); (#202)	
		pthread_mutex_lock (&foodlock); (#274) (v=free)	
		pthread_mutex_unlock (&foodlock); (#291) (v=lock)	
		return of food_on_table function; (#202)	
		pthread_mutex_lock (&fork1); (#192) (v=free)	
pthread_create (D.3881, 0B, philosopher3, 0B); (#120)			
		D.3910 = food_on_table (); (#244)	
		pthread_mutex_lock (&foodlock); (#274) (v=free)	
		pthread_mutex_unlock (&foodlock); (#291) (v=lock)	
	D.3892 = food_on_table (); (#160)		
	pthread_mutex_lock (&foodlock); (#274) (v=free)		
	pthread_mutex_unlock (&foodlock); (#291) (v=lock)		
	return of food_on_table function; (#160)		
	pthread_mutex_lock (&fork3); (#150) (v=free)		
		return of food_on_table function; (#244)	
		pthread_mutex_lock (&fork2); (#234) (v=free)	

wu-ftpd (v2.6.2) is a FTP server software for Unix systems. Up until early 2000s, it was the most common FTP server software in use.

lighttpd (v1.4.19) is light web-server that has been designed for high-performance environments. It has also a low memory footprint.

Data about Petri nets generated from these three programs is provided in table 6.

Table 6. Modeling results for some UNIX programs

	gzip	wu-ftpd	lighttpd
Program's size (lines)	7 788	18 405	52 336
Model ⁶	842/1 119/2 406	4 132/5 331/1 1754	3 403/4 264/8 399
Optimized model ⁶	149/165/498	829/963/3 018	673/761/2 392

8 Conclusion

In this paper, we have presented *Evinrude*, a tool that translates a C program into colored Petri nets 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.

⁶ In terms of places/transitions/arcs

Quickly prototyping Petri nets tools with SNAKES

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 major 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 approach relies on the *perspective* notion. A perspective is a way to aggregate transformation techniques dedicated to a purpose (i.e. system calls, synchronization, etc). Perspectives can be elaborated and adapted according to a new purpose and provides flexibility for program analysis. Perspectives may also be used separately (to focus on one aspect to be analyzed). Finally, they can be composed when several aspects of a program must be analyzed simultaneously (because they interact).

So, *Evinrude*'s transformation process relies on rules 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 [26].

The way we can create, select and compose perspectives in *Evinrude* allows one to control the complexity of specifications produced from source code.

References

1. Krawczyk, H., Wiszniewski, B.: Analysis and Testing of Distributed Software Applications. Taylor & Francis, Inc., Bristol, PA, USA (1998)
2. Clarke, E., Wing, J., et al.: Formal methods: state of the art and future directions. ACM Computing Surveys **28**(4) (1996) 626–643
3. Gogen, J., Luqi: Formal methods: Promises and problems. IEEE Software **14**(1) (1997) 75–85
4. Brim, L.: Parallel model-checking. ERCIM news, special section on Automated Software Engineering **58** (July 2004) 35
5. Edelkamp, S., Leue, S., Lluch-Lafuente, A., Visser, W.: Dagstuhl Seminar on Directed Model Checking (April 2006)
6. Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., McKenzie, P.: Model Checking. In: Systems and Software Verification: Model-Checking Techniques and Tools. Springer Verlag (2001) 39–46
7. Holzmann, G.: An Overview of PROMELA. In: The SPIN model checker. Addison-Wesley (2004) 33–72
8. Girault, C., Valk, R.: Petri Nets for Systems Engineering. Springer Verlag - ISBN: 3-540-41217-4 (2003)
9. Debar, H.: An introduction to intrusion-detection systems. In: Proceedings of Connect'2000, Doha, Qatar, April 29th-May 1st, 2000. (2000)
10. Kordon, F., Voron, J.B., Iftode, L.: Rapid Prototyping of Intrusion Detection Systems. In: Proceedings of the 18th International Workshop on Rapid System Prototyping, Porto Alegre, Brazil, IEEE Computer Society (2007) 89–96

11. Binkley, D.: Source code analysis: A road map. In: FOSE '07: 2007 Future of Software Engineering, Washington, DC, USA, IEEE Computer Society (2007) 104–119
12. Bush, W.R., Pincus, J.D., Sielaff, D.J.: A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.* **30**(7) (2000) 775–802
13. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, New York, NY, USA, ACM Press (2002) 234–245
14. Cole, B., Hakim, D., Hovemeyer, D., Lazarus, R., Pugh, W., Stephens, K.: 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, New York, NY, USA, ACM Press (2006) 673–674
15. Dwyer, M.B., Hatcliff, J., Robby, R., Pasareanu, C.S., Visser, W.: Formal software analysis emerging trends in software model checking. In: FOSE '07: 2007 Future of Software Engineering, Washington, DC, USA, IEEE Computer Society (2007) 120–136
16. Clarke, E., O.Grumberg, Peled, A.: *Model Checking*. MIT Press (2000)
17. Havelund, K., Pressburger, T.: Model checking java programs using java pathfinder. *STTT* **2**(4) (2000) 366–381
18. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: ICSE '00: Proceedings of the 22nd international conference on Software engineering, New York, NY, USA, ACM Press (2000) 439–448
19. Holzmann, G.J., Smith, M.H.: A practical method for verifying event-driven software. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 597–607
20. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. *SIGPLAN Not.* **36**(5) (2001) 203–213
21. Godefroid, P.: Software model checking: The verisoft approach. *Formal Methods in System Design* **26**(2) (2005) 77–101
22. Hillah, L., Kordon, F., Petrucci, L., Trèves, N.: PN standardisation : a survey. In: International Conference on Formal Methods for Networked and Distributed Systems (FORTE'06), Paris, France, IFIP (September 2006) 307–322
23. Move-Team: The CPN-AMI Home page, <http://www.lip6.fr/cpn-ami> (2006)
24. GreatSPN: The GreatSPN Home page, <http://www.di.unito.it/~greatspn>
25. Tip, F.: A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands (1994)
26. Haddad, S.: 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, London, UK, Springer-Verlag (1990) 209–235
27. Dijkstra, E.W.: Hierarchical ordering of sequential processes. *Acta Inf.* **1** (1971) 115–138
28. Microsystem, S.: Sun studio express - thread analyzer readme
29. Varpaaniemi, K.: Prod: An advanced tool for efficient reachability analysis. <http://www.tcs.hut.fi/Software/prod>