Annals of DAAAM for 2010 & Proceedings of the 21st International DAAAM Symposium, Volume 21, No. 1, ISSN 1726-9679
ISBN 978-3-901509-73-5, Editor B. Katalinic, Published by DAAAM International, Vienna, Austria, EU, 2010
Make Harmony Between Technology and Nature, and Your Mind will Fly Free as a Bird
Annals of DAAAM International

INTERPRETING PETRI NETS WITH CLIPS FOR SOFTWARE MODEL CHECKING

BLASKOVIC, B[runo] & RANDIC, M[irko]

Abstract: *In this paper translation of P/T Petri net into the CLIPS expert system shell interpreter has been described. PN translation is the part of software model checking and can serve as analysis tool for finding counterexamples, error trails and for Petri net simulation. At the end an example is presented.*
Key words: *Petri net, CLIPS, expert systems, software model checking, model transformation*

1. INTRODUCTION

The application of formal methods to software verification yields improvements within software development cycle. The methods for software verification based on Software Model Checking (SMC) are nowadays getting more attention from software developers. Although testing is still the most important part of software development, issues like automatic testing, requirement or specification verification can now be reached through model checkers.
In this paper Petri net has been used as model and CLIPS as a tool for analysis. This paper is organized as follows: after Introduction in Section 1, scope and motivation with related work is given in the Section 2. Section 3, 4 and 5 are dealing with tool chain for SMC, CLIPS interpreter and an example, respectively. At the end, there is Conclusion and some further research directions.

2. SCOPE AND MOTIVATION

Model checker takes the model of "object of interest" and "checks" it against desired property expressed by means of logic formula. If logic formula is not true, there is the error in the model, and model checker should produce counterexample with the trail pointing to the error. Natural candidates for modeling are critical fragments of software source code, requirements or specification. In this point several problems arise: the language for model definition is almost always different from the real problem. Even the more, requirements and specifications are given in semi-formal way. Besides that, model checkers can only produce finite representation of model behavior due to the problem known as 'state explosion'. Industrial strength model checkers are capable for model checking within reasonable practical limits.
In this paper we start from Petri net representation of software system, methods for model definition (or model extraction) are out of the scope of this paper. We shall describe the interpreter for *1-bounded* P/T Petri net (PN in subsequent text) realized with CLIPS expert system shell tool. PN has been chosen because PN is more expressive than program's **C**ontrol **F**low **G**raph (CFG) or automata, so PN can be considered as more expressive CFG. After CFG has been transformed to PN, PN model can be analyzed through simulation or model checking. The most important argument for PN and CLIPS approach is error trail analysis. We found CLIPS interpreted PN suitable to produce deep insight of error mechanisms.

2.1 Related work

Here we mention some works based on CFG analysis.
Model checking of C language source code is introduced in (Harris et al., 2010). First, source code is converted into the CFG. During analysis, each path from CFG is examined in order to find property violations. To handle state expansion problem, because loops and similar structures can generate millions of paths, CFG is converted into the maximal strongly connected components. Satisfiability modulo theories with proof based learning engine algorithm is the last step of this approach. Although this approach has been designed for C language, any kind of specification based on finite discrete automata or Petri net can be analysed through model checking.
SPIN model checker is an example of industrial relevant model checking tool, designed for concurrent software verification. In (Holzmann, 2000) each Promela process (Promela is the language of model checker SPIN) can be seen as CFG. Even the more, SPIN has *"built in"* simulation (spin –i <CFG model>) and guided simulation options with modelling constructs like LTL temporal formula and local property assert(<property>) instructions.

3. PN AND SOFTWARE MODEL CHECKING

In software model checking (SMC), model is built directly from the source code. We assume that Petri net represents the model of the software. Automatic translation of source code is still challenging problem. Another challenge is the translation of software formal specification to the model. The whole procedure is summarized on Figure 1. Each number describes step in the sequence of model transformations:

Thus we have: (1) source to CFG, (2) CFG to Petri net, (3) Petri net to CLIPS program, (4) unfolded PN with counterexamples.
In this paper we focus our attention on step (3): Petri net to CLIPS transformation. The first step in software formalization is CFG. CFG is directed graph G(V,E) where:

V- is the set of vertices (nodes), models *values* of the source program variables
E- is the set of edges (arcs connecting nodes). Each edge consists of one or more instructions. Each edge matches transition in PN.

Step (2) from Figure 1. has been introduced because PN has more suitable theoretical background and higher expressiveness then CFG.

$$src \xrightarrow{1} CFG \xrightarrow{2} PN \xrightarrow{3} clp \xrightarrow{4} PN_{unfold}$$

Fig. 1. SMC sequence of transformations

4. CLIPS INTERPRETER FOR P/T NETS

We do not present here detailed or formal description of PN behavior semantic or firing rules, for an overview see (Murata, 1989).
CLIPS (***,1993) has three paradigms: imperative, object oriented and rules. Because of that, CLIPS can serve as natural choice for experiments with PN and declarative programming.
PN can be formally described as mathematical structure consisting of several sets. Such mathematical structure is PN declaration. Each element from PN declaration i.e each element

from the sets has the representation within the CLIPS constructs.

First, we must distinguish between the PN "firing rules" and CLIPS "rules execution" respectively. The transition in PN is enabled when all input places to the transition have one or more tokens. In CLIPS, rule execution depends on agenda. Agenda is the list of all enabled rules in CLIPS program. If more than one rule is on agenda (i.e. enabled), the rule is chosen according to the one of predefined strategy. There is no full compatibility between PN firing rules and CLIPS firing rules agenda, they are not semantically equal. However, this can be solved with additional constraints implemented in CLIPS code. CLIPS interpretation of PN is defined with mapping between PN transition enabling (PN firing rules) and CLIPS firing rules.

4.1 P/T net

Formal PN definition contains enough information to interpret it with CLIPS program. PN is a tuple $PN_{P/T} = (P,T,F,W,M_0)$:

$P = p_1,p_2 \ldots p_n$ is set of places,
$T = t_1,t_2 \ldots t_n$ is set of transitions,
$F \subseteq (P \times T)\ (T \times P)$ is flow relation,
$W : F \rightarrow N\backslash\{0\}$ is arc weight and
$M_0 : P \rightarrow N$ is the initial marking.

Next step is PN formal definition refinement with well known structures from "planning domain description language" (PDDL), each part matches CLIPS construct:

objects	PN states (S), transitions (T), arcs (F)
predicates	properties of objects expressed through LHS in CLIPS rules (example: Is number of tokens in Place x greater than zero?)
actions	execution or PN `token game`, implemented in RHS of clips rules (example: Increment output and decrement input places of transition x)
initial state	PN initial marking M_0
goal	PN final marking, liveness or safe
specification	property

The example of CLIPS code that interprets PN, is presented in the next section.

5. P/T NET EXAMPLE

On Figure 2. part of ABP protocol (Merlin, 1979) is shown.

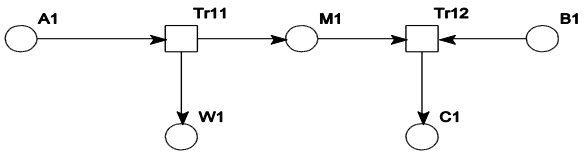

Fig. 2. Part of ABP used in the example

All Places have capacity C bounded to value k=1, "`deftemplate`" defines place as ordered pair with typed values for Place *symbol* and *marking*:

```
(Deftemplate Place "Place for P/T"
 (slot  P     ; for example A1
(type SYMBOL))
 (slot  m     ; marking
(type INTEGER); with
(default 0))) ; default 0
```

PN Places with initial marking are introduced with `deffacts` construct:

```
(deffacts ABP
  (Place (P A1) (m 1))(Place (P M1) (m 0))
  (Place (P B1) (m 1))(Place (P W1) (m 0))
  (Place (P C1) (m 0))…..
```

Transition Tr11 will fire only and only if input marking in Place *A1* is equal to one. LHS finds current markings values and RHS decrements input places and increments output places, respectively. The LHS side of the rule has (a) label and RHS side of he rule has (b) label, respectively.

```
(defrule Tr11
"Transition 11 send message 0"
(a) ?pA1 <- (Place (P A1)(m ?mA1 ))
(a) ?pM1 <- (Place (P M1)(m ?mM1 ))
(a) ?pW1 <- (Place (P W1)(m ?mW1 ))
(a) (test (and (> ?mA1 0) (<= ?mA1 ?*k*)))
 =>(b) (modify ?pA1 (m (- ?mA1 1)))
   (b) (modify ?pM1 (m (+ ?mM1 1)))
   (b) (modify ?pW1 (m (+ ?mW1 1)))
```

5.1 Output

PN execution run is started with CLIPS usual batch routine consisting of `(clear)(load <PNfile>),(reset)` and `(run)` producing the following output:

```
   CLIPS> ==> Tr11: f-1,f-2,f-4
```

Transition `Tr11` is enabled for firing. Fact list `f-1,f-2` and `f-4` are facts representing input and output places. Before firing markings for transition `Tr11` are:

```
   === Tr11 pA1=1 pM1=0 pW1=0
```

After transition `Tr11`, transition `Tr12` is ready for firing:

```
   === Tr12 pM1=1 pB1=1 pC1=0
CLIPS> ==> Tr12: f-18,f-3,f-5
```

and in a similar way other transitions are enabled:

```
  ==> Activation 0      Tr13: f-22,f-7,f-8
  ==> Activation 0      Tr14: f-19,f-24,f-6
```

Note that outputs labeled with `===` are parts of an error trail within PN analysis. With additional CLIPS rules we can analyze desired error trails or explore counterexamples behavior, because LHS of rules contain predicates over PN markings.

6. CONCLUSION AND FURTHER RESEARCH

We have presented CLIPS counterexample analysis tool for SMC realized with CLIPS interpreted P/T Petri Net. With proposed approach we find more detailed elaboration of error trails. In our approach CFG is extended with PN and analyzed with CLIPS. Such approach provides clean and smooth research direction for high level Petri Net interpretation as well as temporal logic formula analysis.

With manual model definition, SMC is still challenging research direction that needs more elaborated model extraction procedures. Another problems exercised are concurrent PN transition firing. We expect that better logic formula for symbolic PN execution should improve PN interpreter.

Future research directions should adjust CLIPS rules as "pure" PN firing rules, interpret high level PN, connect temporal logic formula with CLIPS PN interpreter and introduce automatic test generation. Another research direction should be synthesis: starting from UML or SDL, the goal is the generation of verified software, avoiding model extraction, abstraction and low level code analysis.

7. REFERENCES

Harris, W. R., Sankaranarayanan, S., Ivancic, F., Gupta, A. (2010). Program analysis via satisfiability modulo path programs, pp. 71—82, In: *POPL 2010*

Holzmann, G. (2000). Software Model Checking. Vol. 180 of NATO Summer School, *IOS Press Computer and System Sciences*, pp. 309-355, Marktoberdorf Germany, Aug.2000

Merlin, P. M.(1979). Specification and validation of protocols, *IEEE Transactions on Communications*, pp. 1671-1680, Vol. COM-27, No. 11, November 1979

Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 541--580, Vol. 77, No. 4, April 1989

NASA Software Technology Branch (1993). CLIPS Reference Manual. *Lyndon B. Johnson Space Center*, JSC-25012 Edition, June 1993