A Modular Representation of Fluid Stochastic Petri Nets

Fernando Barros
Universidade de Coimbra
Departamento de Engenharia Informática
3030 Coimbra, Portugal
barros@dei.uc.pt

ABSTRACT

In this paper we develop a modular representation of Fluid Stochastic Petri Nets (FSPNs) using the Hybrid Flow Systems Specification (HFSS), a formalism that combines the concepts of sampling and discrete events to describe hybrid systems. We show that HFSS provides a sound representation of FSPNs supporting a direct mapping between FSPNs elements and HFSS components. FSPNs can be modeled by a composition of HFSS components preserving the structure of the original FSPNs, removing the need for a model transformation layer to simulate FSPNs, or making it easy to develop such a mapping mechanism. We show that the continuous flow representation used by HFSS enables an efficient simulation of FSPNs. Simulation results are presented for a simple manufacturing system with machines subjected to breakdowns.

Author Keywords
Fluid Stochastic Petri nets; modular representations; HFSS formalism; hybrid systems.

ACM Classification Keywords
I.6.5 SIMULATION AND MODELING: Model Development - Modeling Methodologies

INTRODUCTION

Petri Nets (PNs) have widely been used in modeling and analysis of systems. Since their creation many extensions have been developed, including, for example, Timed Petri Nets (TPNs), and Fluid Stochastic Petri Nets (FSPNs) aimed to model timed hybrid systems exhibiting both discrete and continuous elements [11]. In this paper we develop a modular representation of FSPNs using the Hybrid Flow System Specification Formalism (HFSS) [2]. HFSS combines both continuous [1], and discrete (event) flows [13], to represent hybrid systems. We develop a library of HFSS components to represent the elements of a FSPN. These elements are described using the HFSS-Groovy toolkit and include discrete places, transitions and continuous places. We introduce a conflict manager component to explicitly represent transition tie-breaking rules, enabling the use of application dependent algorithms to choose among conflicting transitions. We use infinite server semantics enabling an arbitrary number of transitions to fire simultaneously. The library of HFSS components permits FSPNs to be represented by a structural equivalent HFSS network, that can be obtained through composition using simple transformation rules, in a direct mapping. Structure preserving makes this conversion a very simple process that can be performed manually, also making it easier to define conversion tools, avoiding namely the (costly) compilers that are common in Domain Specific Languages (DSLs) approaches [7]. We present simulation results for a simple manufacturing system with machines subjected to breakdowns. Our results show that HFSS representation of continuous systems by HFSS continuous flows enable an efficient simulation of FSPNs.

TMS/DEVS 2015, April 12-15, 2015, Alexandria, VA, USA

FLUID STOCHASTIC PETRI NETS

FSPNs introduce continuous marking for supporting a representation of hybrid systems [11]. The continuous places of PNs are described by constant rate differential equations enabling a fluid approximation of systems with a large number of tokens. We present next an informal description of FSPN semantics.

Discrete TPNs

Time was introduced in Petri Nets to model system delays. The time to complete a task or a delay in the system can usually be modeled by a stochastic distribution. Timed Petri Nets (TPNs) define a set of transitions, places and arcs. A transition checks its preconditions that depend on the marking of its input places. If preconditions are satisfied, a set of tokens is removed from the input places. After the transition, tokens are added to transition output places. In this paper we assume that time elapses inside transitions [10], departing from the more common TPNs where time elapses in places or arcs. In this paper we also assume infinite server semantics, that allows many transitions to start simultaneously as long as their preconditions are satisfied. The semantics of TPNs can be described by Figure 1 that depicts the behavior of a TPN with transition t_0 and places $p_0, ..., p_3$. The transition precondition requires two units of p_0, one unit of p_1 and it imposes an inhibitor arc of two units of p_2. When the precondition is satisfied a token representing an activity (transition instance) is created within the transition to model a time advance (delay). After this time interval, the transition finishes, and new tokens are created in the corresponding output places. Given the initial marking of Figure 1a, t_0 can start two activities,

Figure 1b. Transition t_0 removes all tokens from p_0 and p_1 and creates two time events to signal the end of the scheduled activities.

Although transition marking is commonly omitted in Petri net analysis, a PN simulator needs to consider it. A standard marking representation could also be used, since a time-in-transition PN can be mapped into a time-in-place PN [10]. The non-standard representation introduced here simplifies simulator description.

When an instance of t_0 finishes the execution it creates new tokens in its output places, Figure 1c, in this Petri net, 1 unit of p_2 and 3 units of p_4. The final marking after the firing of the second instance of t_0 is depicted in Figure 1d. We have assumed that each transitions instances were assigned to a different time duration. If t_0 had a fixed processing time, the two instance would have finished at the same time, jumping the intermediate step of Figure 1c.

Continuous Flows

Fluid Stochastic Petri Nets (FSPNs) were introduced to enable the description of systems requiring a large number of tokens, since an explicit representation of each token would make the PN difficult to analyze and also time consuming to simulate. A fluid approximation is used, and tokens are represented by a real number whose value is governed by a piecewise constant rate.

Before detailing the semantics of FSPNs we define $|t_k|$ as the number of instances of transition t_k currently active. Similarly the quantity (integer or real) of tokens in place p_k if given by $|p_k|$.

In FSPNs, a transition t_k is considered active iff $|t_k| > 0$. When a transition is active the corresponding flow is enabled and place content is influenced by that flow. On the contrary, when not active the corresponding flow is zero. Another constraint imposed is that places can only contain positive values, i.e., $|p_k| \geq 0$.

Figure 2 represents a FSPN with transitions t_0, t_1, t_2, place p_0, constant flows a and b, and a variable flow controlled by the number of tokens in transition t_2.

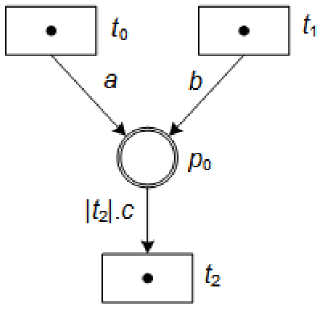


Figure 2 . Continuous flow Petri net with variable rate $|t_2| \cdot c$.

When all the transitions are enabled the content of place p_0 is described by:

$$\frac{\mathrm{d}|p_0|}{\mathrm{d}t} = a + b - |t_2| \cdot c$$

When a transition is disable, the corresponding flow is zero. For example, when $|t_0| = 0$, then $\frac{\mathrm{d}|p_0|}{\mathrm{d}t} = b - |t_2| \cdot c$

Hybrid Flows

FSPNs enable the representation of systems with both discrete and continuous semantics. The FSPN of Figure 3 [8], models a manufacturing system with N machines that process at an (exponential) rate μ and breakdown at an (exponential) rate λ. Entities enter the system at rate a and are processed at rate $|t_1| \cdot d$. The initial number of entities to be produced is given by L and all machines are initially available, $|t_1| = N$. Given the semantics defined before, $|t_1|$ represents the number of machines available for production and $|t_2|$ is the number of machines being repaired (not working).

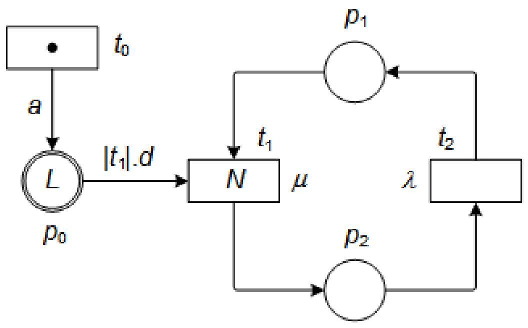


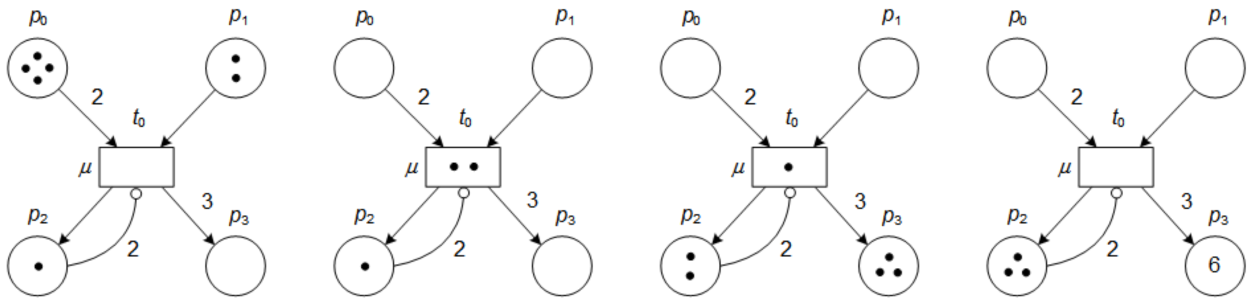
Figure 3 . Petri net with hybrid flows (FSPN) [8].

We have described the main elements of FSPNs. In the next section we provide their representation in the HFSS formalism.

MODULAR REPRESENTATION OF FSPNS

The mapping of FSPNs into a deterministic modeling and simulation formalisms has several advantages. It establishes FSPNs semantics, since modeling formalism like HFSS have deterministic semantics [3]. Modularity enable also the composition of systems from simple elements. Given a FSPN model library developed in HFSS, we can create complex FSPNs by simple composition of basic elements without the need to develop a compiler to generate new HFSS models from a FSPN specification, a requirement usual in Domain Specific Languages for representing (non-timed) PNs [7]. Additionally, mapping FSPNs into a modeling formalism can also exploit the advantages of existing and efficient simulation kernels without the need to create a specific solution for FSPNs.

In this paper we use the Hybrid Flow System Specification (HFSS) formalism for modeling and simulating FSPN. HFSS is a hierarchical and modular formalism and we developed a representation PN targeting a one-to-one mapping between PN elements (places, transitions, ...), and HFSS components. In this manner we can map a PN into a composition of HFSS models. Our goal is to achieve a trivial mapping between PNs and HFSS networks, so a simple (and possibly automatic) translation can be achieved. In this manner we avoid generating HFSS basic models from PNs, limiting the translation task to the composition of per-defined HFSS models.



(a) initial marking (b) fire of transition t_0 (c) end of processing of a token in t_0 (d) final marking

Figure 1 . Discrete Petri net evolution.

Other approaches, like Domain Specific Languages (DSLs) approaches, that require a special purpose compiler to map PN into a modeling formalism, becoming more time consuming. Additionally, our approach keeps the structure of the PN, enabling further refinements. This flexibility, allows, for example, to combine HFSS components used in PNs with other HFSS models becoming an effective manner to obtain very expressive approaches. On the contrary a code generation approach would map a PN into a monolithic atomic model in some modeling formalism making it, in our opinion, difficult to understand and to modify/maintain.

The HFSS formalism combines several abstractions, including adaptive sampling, continuous flows, and discrete events. HFSS models are modular communicating through a well defined interface. A HFSS model can read and produce continuous and discrete flows (events), offering a framework for defining hybrid models [3]. The HFSS-Groovy toolkit is a Groovy language implementation of the HFSS formalism and it is used in the next sections to describe the HFSS components required to represent FSPNs.

Discrete Places

We start by describing a HFSS model of FSPNs discrete places. This model requires the ability to change its content supporting the basic discrete event operations of adding and removing tokens, and to communicate changes in this value. Since a key operation in a FSPN is to test preconditions we provide the access to the place current number of tokens through the component continuous output flow function. The HFSS `Place` is represented in Figure 4.

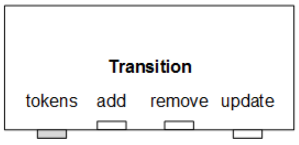


Figure 4 . HFSS discrete `Place` model.

HFSS-Groovy definition of the `Place` model is given in Listing 1. Class `Model` (not detailed here), provides the basic support to HFSS and it defines variable `alpha` and `beta` to set the time to read (sample), and the time to write (produce a discrete flow). A `Place` is created with an initial number of `tokens` and it becomes passive (`alpha` = `beta` = ∞), waiting for an input.

```
public class Place extends Model {                                        1
  private int tokens;                                                     2
  public Place(String name, int tokens) {                                 3
    super(name);                                                          4
    this.tokens = tokens;                                                 5
    alpha = Double.POSITIVE_INFINITY;                                     6
    beta = Double.POSITIVE_INFINITY;                                      7
  }                                                                       8
  public void transition(double e, def xc, def xd) {                      9
    beta = Double.POSITIVE_INFINITY;                                      10
    if (xd == null) return;                                               11
    int prev = tokens;                                                    12
    xd.at("add").each {Port p-> tokens += p.value()}                      13
    xd.at("remove").each {Port p-> tokens -= p.value()}                   14
    if (prev != tokens) beta = 0;                                         15
  }                                                                       16
  public def outputC(double e) {return new Port("tokens", tokens)}        17
  public def outputD(double e) {return new Port("update", tokens)}        18
}                                                                         19
```

Listing 1 . HFSS-Groovy `Place` model.

A `Place` receives commands to change its content in ports `add` and `remove`. Since HFSS uses a parallel semantics, an input is usually a list of pairs in the form (port name, value). The transition function (line 9), specifies the behavior of `Place` input arrival. Each `add` increases the number of tokens, and each `remove` decreases this value. When the number of tokens is modified the updated value is sent through the discrete port `update`, as defined by function `outputD` (discrete output). The current number of tokens is always available through the continuous output flow port `tokens`, as specified by the function `outputC` (continuous output). This continuous value plays a key role in simplifying the definition of the HFSS model of a FSPN as we show in the next sections. While the input is discrete, the output has a (piecewise constant) continuous flow with the current number of tokens, and a discrete flow trajectory, signaling a change in this number. Typical `Place` trajectories are shown in Figure 5.

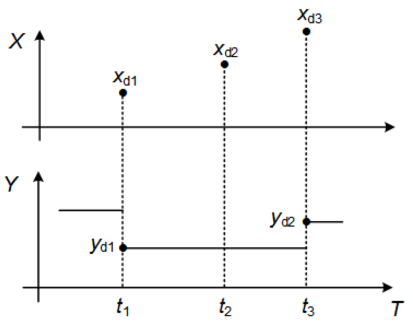


Figure 5 . HFSS `Place` trajectory.

Continuous/Fluid Places(Reservoirs)

When the number of tokens in a FSPN is very large a fluid approximation simplifies the analyses and enables a more efficient simulation. In this approximation transitions are continuous and become characterized by the rate they modify reservoir (fluid places) contents. The HFSS model of a reservoir in given in Figure 6.

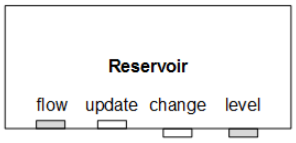


Figure 6 . HFSS `Reservoir` model.

The `Reservoir` samples the input flow and integrates this value, that is constrained to be positive. Given a negative input rate, reservoir level decreases until it reaches zero, and keeps this value irrespective to a negative input rate. Since, FSPNs constrain input rates to be piecewise constant, fluid integration involves only transition when these rates changes. Input port `update` receives a signal when the rate is modified. The current rate is available at input port `flow`. The reservoir defines also the continuous output port `level`, to provide access to the current reservoir level, and the discrete port `change` to signal a modification in the reservoir flow rate. The HFSS-Groovy implementation is described in Listing 2. Where method `rate`, line 10, computes the effective input rate, and method `level`, line 34, computes reservoir contents. When the rate is zero, line 28, the model becomes passive.

```
class Reservoir extends Model {                                        1
    private double level;                                              2
    private double rate = 0.0;                                         3
    public boolean isZERO(double x) {return Math.abs(x) <= 1.0e-12}    4
    public Tank(String name, double level) {                           5
        super(name);                                                   6
        this.level = level;                                            7
        alpha = 0.0;                                                   8
    }                                                                  9
    private double rate(double r) {                                    10
        if (isZERO(r)) return 0;                                       11
        if (isZERO(level) && r < 0) {                                  12
            level = 0;                                                 13
            return 0;                                                  14
        }                                                              15
        return r;                                                      16
    }                                                                  17
    public void transition(double e, def xc, def xd) {                 18
        this.passivate();                                              19
        level = level(e);                                              20
        double nextRate = rate(xc.value());                            21
        if (! isZERO(nextRate - rate)) {                               22
            rate = nextRate;                                           23
            beta = 0.0;                                                24
            return;                                                    25
        }                                                              26
        rate = nextRate;                                               27
        if (isZERO(rate)) return;                                      28
        if (rate < 0) {                                                29
            beta = -level / rate;                                      30
            return;                                                    31
        }                                                              32
    }                                                                  33
    private double level(double e) {return level + rate * e}           34
    public def outputC(double e) {return new Port("level", level(e))}  35
    public def outputD(double e) {return new Port("change", level(e))} 36
}                                                                      37
```

Listing 2 . HFSS-Groovy `Reservoir` model.

Typical reservoir input and output trajectories are depicted in Figure 7. The input flow changes at instants t_1, t_2 and t_4. This value is sampled from the continuous input trajectory and imposed by the arrival of discrete flows. Tank flow level is continuous, a piecewise linear flow, as a consequence of the piecewise constant flow rate constraint. A discrete flow signal is produced each time the flow changes. In the interval $[t_3, t_4]$ the reservoir content is zero due to the negative input rate.

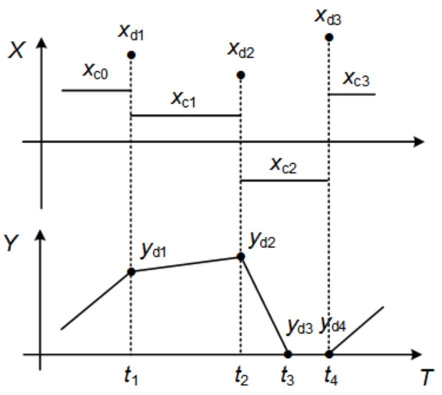


Figure 7 . HFSS reservoir trajectories.

Transitions

HFSS transition model is represented in Figure 8. A transition samples the precondition from the continuous ports `tokens[n]`, that are connected to the input places the transition depends upon. If the precondition is true the transition tries to seize the tokens from the `Conflict Manager` component that manages place access conflicts. When an acknowledged is received the transition executive `Transition`$_\eta$ creates a `Delay` to signal transition end. We assume the infinite server semantics and thus multiple copies of a transitions (instances) can execute in parallel.

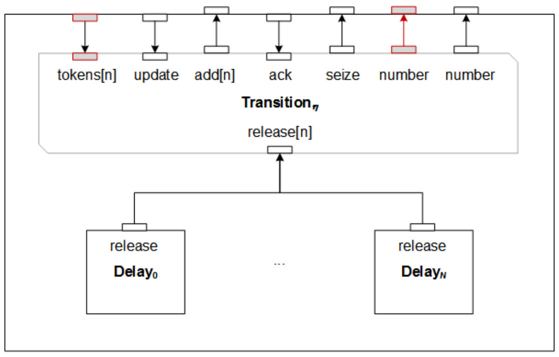


Figure 8 . HFSS transition model.

When a `Delay` finishes it signals the executive that removes it from the network and releases the corresponding tokens through the output ports `add[n]`. A transition also checks for its precondition when receives an `update` message sent by a place that has changed its number of tokens. The current number of `Delay` instances can be sampled at the continuous output port `number`. When this value changes a signal is sent through the discrete output port `number`.

Conflict Manager

Given the infinite server assumption, and since HFSS has a parallel semantics, implying that all transitions scheduled to the same time must be fired simultaneously, conflicts can arise. Conflicts need to be solved in a deterministic manner, and under the control of the modeler. These requirements impose a centralized controller to decide what transitions can be fired and those that need to hold. The `ConflictManager` model is represented in Figure 9. It receives requests in port `seize` from transitions whose preconditions are satisfied. The information about tokens availability is sampled at input ports `tokens[n]`. At this point the conflict manager has full control on what transitions can proceed. Many strategies to break ties are possible, including a probabilistic rule, fairness considerations, waiting time, priorities, etc. Since we have an explicit representation of conflicts it becomes possible to use any algorithm suitable for a particular goal.

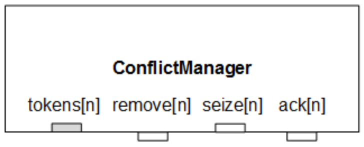


Figure 9 . HFSS conflict manager model.

When the manager decides to enabling a transition it removes the tokens from the corresponding places using output port `remove[n]`. Transition acknowledgment is made through output port `ack[n]`.

We have described the basic HFSS components that can be used to represent FSPNs elements. In the next sections we show how these HFSS components can be combined to create arbitrary FSPNs.

Continuous Flow HFSS Model

We start the HFSS representation of FSPNs by considering the continuous FSPNs of Figure 2. The mapping from the FSPN to the corresponding HFSS network model is straightforward and is depicted in Figure 10.

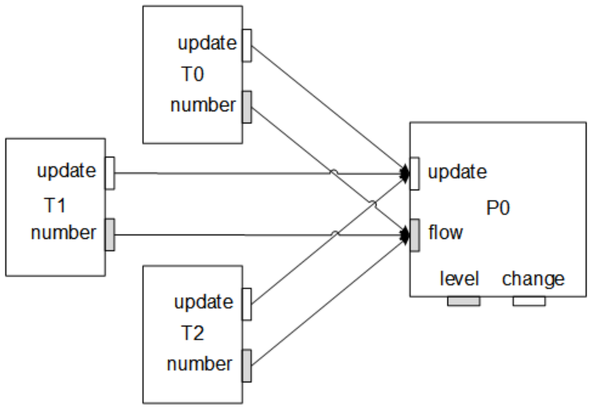


Figure 10 . HFSS continuous flow network model.

Reservoir P0 samples the input rate from transitions T1, T2 and T3, and any change in the number of tokens, forces a sampling operation in P0. Thus P0 input rate is updated whenever there is a change in any input transition. The main difference between the FSPN of Figure 2 is that the link $p_0 \rightarrow t_2$ becomes mapped into a HFSS link $t_2 \rightarrow p_0$. FSPN link direction indicates the fluid flow, while in HFSS the fluid flow needs to be numerically defined as either positive or negative as shown in Listing 3 of the next section.

Discrete Flow HFSS Model

The mapping of the FSPN of Figure 1 into a HFSS model is represented in Figure 11. FSPN places and transitions are mapped into the corresponding HFSS components described before. The conflict manager completes the HFSS model since it is required to handle the access to (common) places.

We can observe that the structure of the original PN is mainly preserved. Transition T0 reads from places P0 and P1 and it produces tokens to places P2 and P3. However, the arity of the arcs is stored in ports input function and it is not explicitly represented in the Figure 11. The conflict manager is introduced to explicitly represent the algorithm for solving collisions when transitions access to the same resources. This element is not used in PN diagrams and need to be specified by some textual annotation. Our approach makes it possible to define different strategies and making them reusable HFSS components that can be chosen according to the system requirements.

SIMULATION RESULTS

For the validation of the HFSS representation of FSPNs we use the petri net of Figure 3, with parameters $L = 90$, $N = 5$, $\mu = 1/4.0$, $\lambda = 1/1.5$, $a = 50$, and $d = 14$. The input rate of

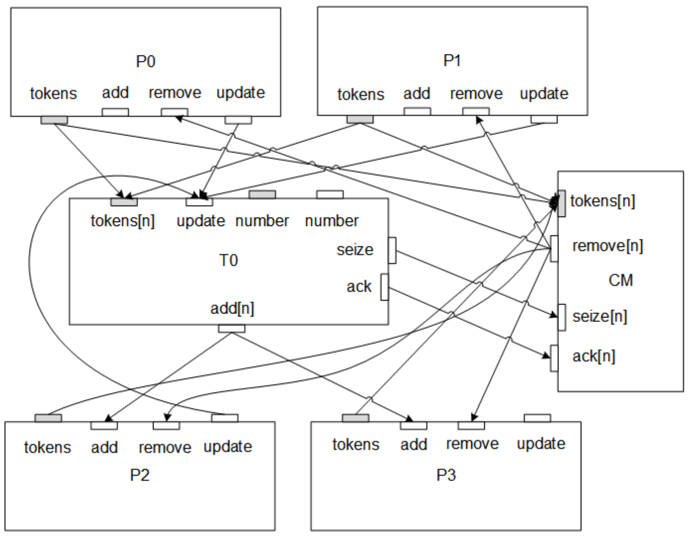


Figure 11 . HFSS discrete flow network model.

p_0 is set by HFSS-Groovy method `influencers` defined by Listing 3.

```
influencers("P0", ["T0", "T1"],
   {List xc->
      def res = xc.filterByPort(["number"]);
      return res[0].value() * 50 - res[1].value() * 14;
   }, {List xd->
      def res = xd.filterByPort(["number"]);
      return res;
});
```

Listing 3 . P0 input function.

Simulation results for the contents of reservoir p_0 are depicted in Figure 12, where discrete flows are represented by squares and the continuous flow is piecewise linear.

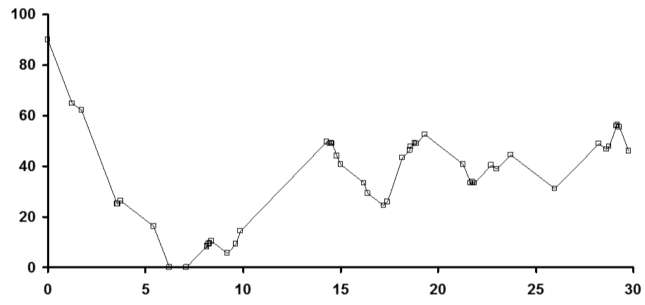


Figure 12 . Marking of place p_0.

As can be observed from the graphic, the number of transitions is very small (#transitions = 53), since the HFSS representation exploits the piecewise constraint of input flows to achieve an efficient simulation. On the contrary, a general propose ODE solver like Adams, Runge-Kutta or quantization would require a higher number of transitions. Given that HFSS models can produce continuous output flows, reservoir continuous trajectory can be calculated without involving any transition, reducing the computation cost.

RELATED WORK

To the best of our knowledge we have developed the first modular description of FSPNs. The representation of Petri Nets in discrete event formalisms, like DEVS, have been described [9]. However, these approaches relate only to non-timed PNs with single server semantics exhibiting very simple requirements when compared to the FSPNs modeled in this paper. Although a DEVS representation of a FSPN could be developed, it would become more complex than the corresponding HFSS representation, due to the lack of sampling semantics in the DEVS formalism. The reuse of DEVS models would also be compromised since the representation of sampling in a discrete event formalism requires the adaption of FSPN models, like places and transitions, to every specific topology [4]. The discrete event simulation of FSPNs has been described in [5]. This work, however, provides a description of an *ad hoc* implementation not based on a modular representation. Implementations based on fixed sampling rates have been developed [6], but results are dependent on the sampling rate, and loose accuracy when compared with the *exact* results achieved by HFSS models. The use of HFSS sampling provides a simplification in the messages required to retrieve information. Taken for example the conflict manager component, it requires one sampling operation to retrieve token information from all connected places. The alternative discrete event representation would require sending a message to each place and then wait for every answer to arrive. HFSS sampling enables all this information to be exchanged in a single atomic operation, being easier to represent, while promoting model reuse. Domain Specif languages (DSLs) have been developed to create simulations from (non-timed) PNs. However, these approaches require the development of specific compilers in order to be applied [7]. On the contrary, our approach based on a domain specific (FSPN) library, represented in Figure 13 makes the mapping between

FSPNs and simulations very simple, removing the need for expensive compilers. If a compiler is required, its creation becomes simplified since it only needs to map FSPNs into a network of HFSS components as described in this paper. Our conjecture is that the approach of Figure 13 can also be applied to other modeling formalisms where HFSS representations (libraries) can be developed.

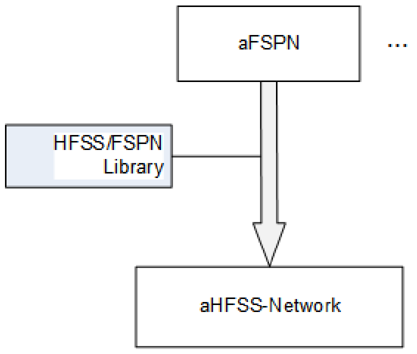


Figure 13 . Library-based model generation.

While modeling formalisms can have a discrete event representation at the atomic level [12], this has little relevance in the description of complex models. The representation, for example, of FSPNs by a composition of basic components is of utmost importance if our aim is to simply the automatic translation of FSPNs into executable simulation models. As shown before, our modular representation based on HFSS makes it very simple to map a FSPN into a HFSS network, requiring no code generation since it is based on the reuse of the predefined components. On the contrary, a solution based on the transformation of FSPNs into one (complex) atomic model will mostly require a compiler to perform the task. Moreover, this equivalent atomic model will become very difficult to be understood by a (human) modeler. Additionally, our modular approach enables FSPNs to be easily combined with components from other domain specific libraries to achieve complex representations based on multiple formalisms.

Another advantage of the HFSS modular representation relates to compiler maintenance and optimization. A compiler producing code form scratch, requires the mapping of FSPNs into one large HFSS atomic model, making it very difficult the future optimizing of the generated code, since transformations would become hard-coded into the compiler. On the contrary, a *light* compiler enabled by our approach, keeps the libraries outside the code generation procedure, greatly simplifying maintenance and optimization operations, that can be made through library improvement without requiring changing the compiler itself.

CONCLUSION AND FUTURE WORK

We have developed a modular representation of FSPNs based on the HFSS formalism that enables a direct map of petri net elements, like places and transitions, into a HFSS components. The HFSS representation preserves the structure of the original FSPN, becoming very easy to develop and amenable to a manual translation. Our approach makes it also possible FSPNs to be defined directly from a library of HFSS components without requiring the support for translation/generation tools. As future work we plan to model Hybrid Petri Nets [6], an alternative to FSPNs with more powerful semantics, namely the ability to describe inhibitor arcs associated with continuous places. The methodology described here for creating HFSS simulation models from FSPNs will also be applied to other formalisms.

REFERENCES

1. Barros, F. Towards a theory of continuous flow models. *International Journal of General Systems 31*, 1 (2002), 29–39.

2. Barros, F. Dynamic structure multiparadigm modeling and simulation. *ACM Transactions on Modeling and Computer Simulation 13*, 3 (2003), 259–275.

3. Barros, F. Semantics of discrete event systems. In *Distributed Event-Based Systems* (2008), 252–258.

4. Barros, F., and Zeigler, B. Model interoperability in the discrete event paradigm: Representation of continuous models. In *Modeling and simulation: Theory and practice*, G. Bekey and B. Kogan, Eds. 2003.

5. Ciardo, G., Nicol, D., and Trivedi, K. Discrete-event simulation of fluid stochastic Petri nets. *IEEE Transactions on Software Engineering 25*, 2 (1999), 207–217.

6. David, R., and Alla, H. *Discrete, Continuous, and Hybrid Petri Nets*. Springer, 2005.

7. Esser, R., and Janneck, J. A framework for defining domain-specific visual languages. In *OOPSLA Workshop on Domain-Specific Visual Languages* (2001).

8. Horton, G., Kulkarni, V. G., Nicol, D. M., and Trivedi, K. S. Fluid stochastic petri nets: Theory, applications, and solution techniques. *European Journal of Operational Research 105* (1996), 184–201.

9. Jacques, C., and Wainer, G. Using the C++ DEVS toolkit to develop Petri Nets. In *Summer Computer Simulation Conference* (2002), 252–258.

10. Ramchandani, C. *Analysys of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD thesis, Massachusetts Institute of Tecnology, 1974.

11. Trivedi, K., and Kulkarni, V. FSPNs: Fluid stochastic Petri Net. In *Application and Theory of Petri Nets, Lecture Notes in Computer Science*, vol. 691 (1993), 24–31.

12. Vangheluwe, H., de Lara, J., and Mosterman, P. J. An introduction to multi-paradigm modelling and simulation. In *AI, Simulation and Planning in High Autonomy Systems* (2002), 9–20.

13. Zeigler, B., Praehofer, H., and Kim, T. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.