8-2013

Mapping of UML Diagrams to Extended Petri Nets for Formal Verification

Byron DeVries
Grand Valley State University

Mapping of UML Diagrams to Extended Petri Nets for Formal Verification

Byron DeVries

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfilment of the Requirements

For the Degree of

Master of Science

Computer Information Systems

August 2013

Abstract

PURPOSE: UML Statechart Diagrams are the industry standard for modeling dynamic aspects of system behavior. However, other behavioral models, such as extended Petri Nets, are significantly easier to analyze formally. This research project creates methods of converting previously unconvertible features of UML Statechart Diagrams to extended Petri Nets to allow for additional analysis of UML Statechart Diagrams. PROCEDURES: Algorithms are introduced that convert specific UML Statechart Diagrams to a novel behavioral construct, Swim Lane Petri Nets, and subsequently to extended Petri Nets. Algorithms are also introduced to convert both Swim Lane Petri Nets and extended Petri Nets to the PROcess MEta LAnguage (PROMELA) to allow for detailed formal verification using the SPIN model checker. OUTCOME: Formal definitions of the behavior models, and algorithms for conversions between the models are presented with a focus on traceability between translated models to allow for backtracking the results of formal analysis in the SPIN model checker to the original behavioral construct. IMPACT: While UML Statechart Diagrams are the industry standard and provide an intuitive representation of behavior models, formal analysis is limited and difficult. Providing a method of translation to extended Petri Nets, which are more analyzable but less intuitive, adds significant practical value the use of UML Statechart Diagrams in model based development.

Contents

List of Figures

List of Algorithms

Part I

Introduction

Software development, especially that of concurrent systems, where bugs can be difficult to reproduce, is a difficult undertaking. One of the giants of computer science, Edsger Dijkstra, said, "Program testing can be used to show the presence of bugs, but never to show their absence."[6, p. ix] It is this very impossibility that drives efforts to further understand, decompose, analyze, and verify software systems. One of the many methods of adding a level of sanity to a difficult task is modeling. By using representations that are limited in some way it is possible to more easily analyze a given problem in this representation. Of course it is not always easy to cajole a problem into any given representation. In fact some representations are much more amenable to modeling certain types of problems - though often the easier it is for a representation to model any given problem the harder it is to analyze formally.

One modeling representation that is rich enough in features that modeling most anything is direct, or more direct than other representations, is UML Statechart Diagrams. UML Statechart Diagrams are the industry standard for modeling dynamic aspects of system behavior. However, other behavioral models, such as extended Petri Nets, are significantly easier to analyze formally. For example, given a Statechart (Figure 1), it is not necessarily obvious which states could affect other states due to the numerous representative features of transitions within Statecharts. However, in a Petri net (Figure 2), it is quite easy - the connections are shown explicitly. If two Petri net places are connected by a transition and a pair of arcs, then it is trivial to see the connection. Certainly it is more obvious in Petri nets that there are two transitions from H to B in the Statechart in Figure 1.

This thesis intends to both create methods of converting previously unconvertible features of UML Statechart Diagrams to extended Petri Nets, but also recreate existing feature transition with a focus towards traceability to allow for additional analysis of UML Statechart Diagrams. Trace-

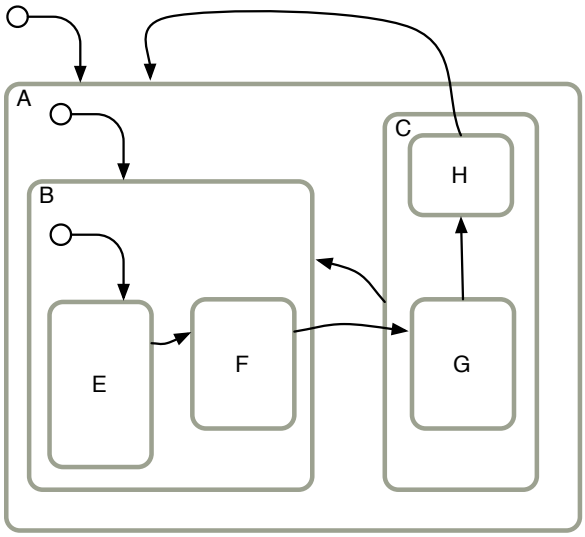

Figure 1: Difficult to Analyze Statechart

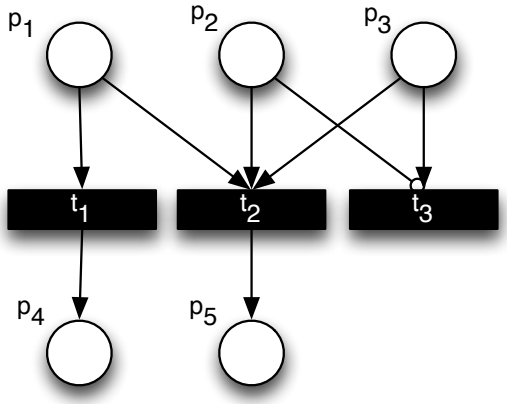

Figure 2: Easy to Analyze Petri Net

ability is a fundamental issue in analysis of translated models. If a representation, say Petri nets, allows for analysis or verification unavailable in another representation, such as Statecharts, but there is no method of applying the results back to the original then there is little value.

To enable the necessary translations between different representations algorithms are introduced that convert specific UML Statechart Diagrams to a novel behavioral construct, Swim Lane Petri Nets, and subsequently to extended Petri Nets. Algorithms are also introduced to convert both Swim Lane Petri Nets and extended Petri Nets to the PROcess MEta LAnguage (PROMELA) to allow for detailed formal verification using the SPIN model checker. Formal definitions of the behavior models and algorithms for conversions between the models are presented to support formal

analysis in the SPIN model checker to the original behavioral construct.

While UML Statechart Diagrams are the industry standard and provide an intuitive representation of behavior models, formal analysis is limited and difficult. Providing a method of translation to extended Petri Nets, which are more analyzable but less intuitive, adds significant practical value the use of UML Statechart Diagrams in model based development.

Part II

Background

This thesis draws upon several different, but highly related areas: models, modeling, and model checking. These areas are covered in this section with overviews of SPIN and PROMELA, Petri nets, and Statecharts.

1 SPIN and PROMELA

1.1 Overview of SPIN Uses

The SPIN model checker is a tool for automating the verification of modeled systems. [1, p. 1] The language that the SPIN model checker uses is PROMELA (Process or Protocol Meta Language). PROMELA code is the goal of the transitions presented in this thesis. All transitions ultimately lead to a PROMELA representation for model checking.

1.2 Overview of PROMELA and SPIN dataflow

The SPIN model checker uses PROMELA source files to either simulate a model or create a verifier of the model represented by the PROMELA source. The method of simulation is direct, that is, the SPIN executable directly simulates the model represented in PROMELA as shown in Figure 3.

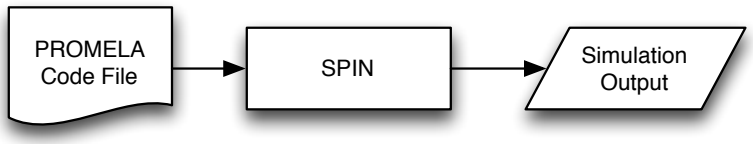

Figure 3: Flowchart of PROMELA simulation with SPIN

However, running a simulation only executes one potential execution path through the model. If there are multiple potential execution paths, one of them is chosen at random. To perform a more

detailed verification of the model that covers more than a single random execution path, SPIN must be used to create a verifier of the program. This verifier is C source code that can then be compiled by a standard C compiler and executed to explore all execution paths, as shown in Figure 4.

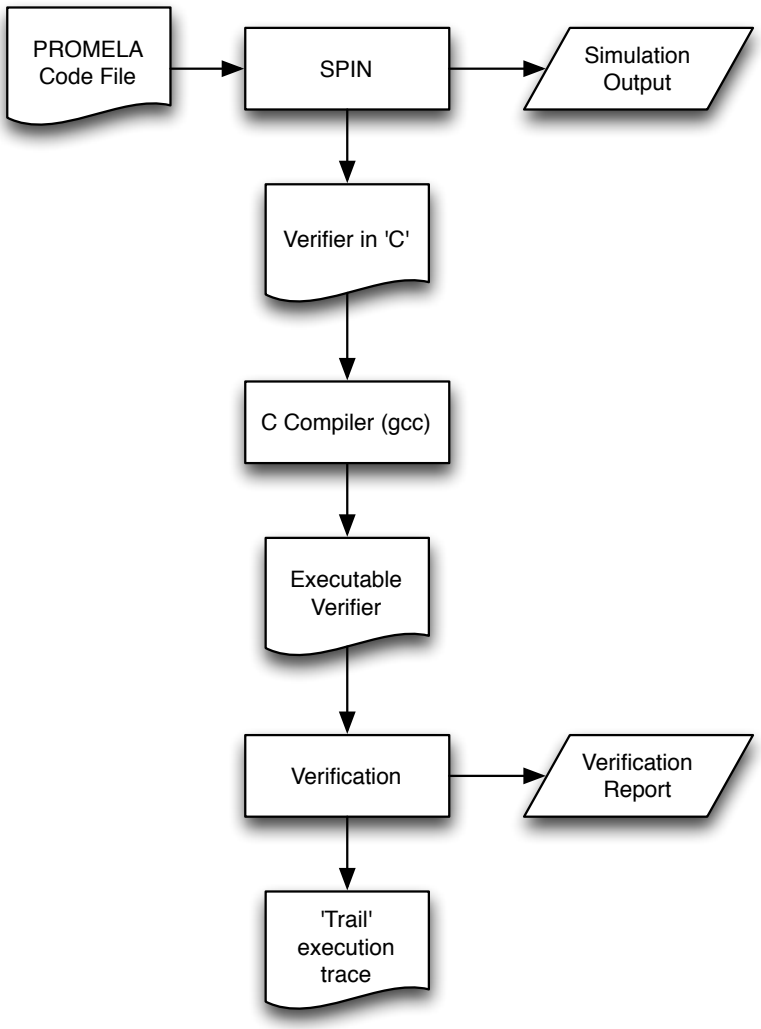

Figure 4: Flowchart of PROMELA verification with SPIN

The execution of this generated verifier can be designated to halt at the first issue found or to continue and attempt to find all potential issues. The detailed information is stored in a '.trail' file indicating the paths of execution.

1.3 PROMELA Language

The PROMELA language for describing models that can be simulated or verified with SPIN is not dissimilar, initially, to other languages and is often compared to the C language.[1, p. 1] Many of the same control and looping constructs are available as well as familiar data types, though most notably floating data types are not available.

1.3.1 Data types

The data types available within PROMELA are divided into two types, numeric and other. The numeric data types are bit, bool, byte, short, int, and unsigned. Their ranges and size are:

Table 1: Data Types

Type	**Values**	**Bits**
bit	0, 1	1
bool	true, false	1
byte	0 to 255	8
short	-32768 to 32767	16
int	-2^{31} to $2^{31}-1$	32
unsigned VAR: x	0 to 2^n-1	1 to 32

The other data types are chan, pid, and mtype. The chan data type is a pre-defined data type for message passing, the mtype type is used to define symbolic names for numeric constants, and the pid type is the type of _pid which is a local read-only variable that stores the process ID for the current process. [1, p. $4-5$]

1.3.2 Operators and Expressions

Operators, and thus similarly expressions are almost identical to the C language, with the following exception: expressions must be evaluable without side effects. This has a few consequences, the first being that assignment does not return the assigned value. It is not possible, therefore, to have an assignment on the right side of an assignment like you could in C.

```
int a = 5;
int b = 10;
a = (b = 0) + 5;
```

Similarly, operators that perform an assignment no longer are valid. Therefore prefix increment and decrement operators do not exist in PROMELA, making `++a` invalid while `a++` is still valid. [1, p. 6 – 7]

1.3.3 Selection Statements

Selection statements are in PROMELA diverge from the C-like constructs that have been presented in PROMELA thus far, both syntactically and behaviorally. In C a conditional `if` statement is written as so:

```
int a = 5;
int b = 10;
if(a == 5) {
    printf("a == 5\n");
} else if(b == 10) {
    printf("b == 10\n");
} else if(a == 5 && b == 10) {
    printf("a == 5 && b == 10\n");
}
```

While in PROMELA the same conditional would be written as:

```
int a = 5;
int b = 10;
if
:: a == 5 -> printf("a == 5\n")
:: b == 10 -> printf("b == 10\n")
:: a == 5 && b == 10 -> printf("a == 5 && b == 10\n")
fi
```

While the code listings illustrate the syntactic differences, there are also behavioral differences. Specifically in the PROMELA example there is only one `if` statement and the conditionals within it are not order dependent. That is, the first true choice is not necessarily the execution path that is

taken. In the case that multiple conditionals are true, only one of the true conditionals execution path will be taken. This means that in the case of the PROMELA code, while every single one of the conditionals could be taken only one will be taken. The output could be any one of the following:

```
a == 5
```

```
b == 10
```

```
a == 5 && b == 10
```

This is different from the C example where only the first conditional will be taken, since it is true, always giving the output:

```
a == 5
```

Even if the C example was changed to not use the `else` statement, as so:

```
int a = 5;
int b = 10;
if(a == 5) {
    printf("a == 5\n");
}
if(b == 10) {
    printf("b == 10\n");
}
if(a == 5 && b == 10) {
    printf("a == 5 && b == 10\n");
}
```

Since every conditional's execution path could be taken, they are all taken, always giving the output:

```
a == 5
b == 10
a == 5 && b == 10
```

This is a significant behavioral difference in how PROMELA and C treat conditional expressions.

1.3.4 Repetition Statements

The only method of repetition (outside of a `goto` statement) in PROMELA is the `do` statement. Similar to an `if` statement the `do` statement is a set of conditionals with actions that are executed until the `break` keyword is executed. For example, this would output the numbers 1 to 10:

```
int N = 10;
int count = 1;
do
:: count <= N -> printf("count = %d", count); count++
:: count > N -> break
od;
```

Just like the `if` statement, the conditionals will randomly execute a single execution path out of the conditionals that are true. Due to that, writing loops in PROMELA can be a bit different than writing loops in other languages.

1.3.5 Concurrency

Concurrency can be modeled in PROMELA by creating processes, whose statements are interleaved. For example, the following is a program listing that executes two output statements in parallel by labeling them both `active`:

```
active proctype A() {
    printf("Output from A\n")
}

active proctype B() {
    printf("Output from B\n")
}
```

The possible output of this code is any one of the following:

```
Output from A
Output from B
```

```
Output from B
Output from A
```

It is also possible to create multiple concurrent instances, as so:

```
active [2] proctype A() {
    printf("Output from A, process = %d\n", _pid)
}
```

Which would produce the following possible outputs:

```
Output from A, process = 0
Output from A, process = 1
```

```
Output from A, process = 1
Output from A, process = 0
```

As mentioned in the data types section, the `_pid` variable can be used to differentiate concurrent processes, even when running the same executable code. It is also possible to leave off the `active` keyword and manually start parallel processes as so:

```
proctype A() {
    printf("Output from A, process = %d\n", _pid)
}

init {
    atomic {
        run A();
        run A()
    }
}
```

Which gives a slightly different output from before:

```
Output from A, process = 1
Output from A, process = 2
```

```
Output from A, process = 2
Output from A, process = 1
```

The `_pid` values are one higher because running the `init` section counts as a process that subsequently started another two processes.

1.3.6 Assertion

Assertions in PROMELA are the method in which verification is accomplished. In either simulating a PROMELA model or verifying one, the assertions check for specific cases that are considered outside of the expected behavior. The `assert` function simply accepts an expression that evaluates to either `true` or `false`. For example, updating a previous example to `assert` any processor use over 2 will correctly fail the assertion:

```
proctype A() {
    printf("Output from A, process = %d\n", _pid);
    assert(_pid < 2)
}

init {
    atomic {
        run A();
        run A()
    }
}
```

This code would work if we did not start the processes from the `init` code block, which adds another process and could easily be overlooked.

1.4 Specifics of SPIN

PROMELA itself is not much use without an engine to run it either as a simulation or for verification. The commands are specific depending on the intended method of execution. In order to run a PROMELA source code file in simulation mode simply call the SPIN executable with the source code file as an argument. For example, if the source file is `example.pml` the command would be:

```
spin example.pml
```

If you wanted to run that same PROMELA source code file in verification mode, the commands would be:

```
spin -a example.pml
gcc -o pan pan.c
./pan
```

First SPIN creates a C source file, which is then compiled to an executable using a C compiler (in this case GCC). That executable can then be executed and any faults are searched for instead of simulating one execution path.[6]

2 Petri Nets

Petri nets are an "accepted model for protocols and event-driven applications" [9, p. 107] while still having formal semantics that "are well suited for formal verification." [2]

2.1 Standard Petri Net

Definition 1. *A* Petri net *is a bipartite-directed graph* (P,T,I,O) *in which P and T are disjoint sets of nodes, and I and O are sets of edges, where* $I \subseteq P \times T$*, and* $O \subseteq T \times P$*. [9, p. 107]*

The places in a Petri Net, or P in the Definition above, are represented as circles while the transitions, or T in the Definition above, are represented as rectangles. The I and O represent the connections between the places and transitions. A visual example of a Petri Net is shown in Figure 5.

This Petri net can also be described mathematically using Definition 1 as a Petri net, C, where:

$$\begin{aligned} C &= (P,T,I,O) \\ P &= \{p_1,p_2,p_3,p_4\} \\ T &= \{t_1,t_2\} \\ I(t_1) &= \{p_1,p_2,p_3\} \\ I(t_2) &= \{p_3\} \\ O(t_1) &= \{p_4\} \\ O(t_2) &= \{p_1\} \end{aligned}$$

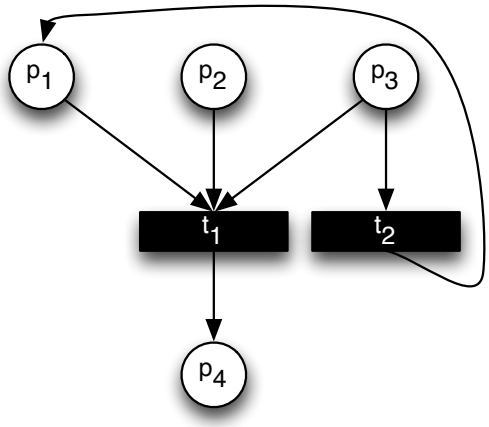

Figure 5: Petri Net Visual Example

2.2 Marked Petri Net

However, even though there is executable logic defined in the way the places and transitions interconnect, there is a need for data that the executable logic acts on. This gives the extension of marked Petri nets, as defined in Definition 2.

Definition 2. *A* marked Petri net *is a 5-tuple* (P,T,I,O,M) *in which* (P,T,I,O) *is a Petri net and* M *is a set of mappings of places to natural numbers. [9, p. 108]*

Places store a number of tokens, initially defined by markings in M. Transitions are enabled if and only if all input places have at least one marking per connection. Of the enabled transitions one is fired, either randomly or by selection, at which point each input place loses one token per connection and each output place gains one token per connection.

2.2.1 Marked Petri Net Execution Example

For example, given the marking $M = (0,1,2,0)$ for the Petri net shown in Figure 5, we have a marked Petri net as shown in Figure 6.

In Figure 7, step 1, t_2 is enabled since all of its inputs (p_3) have tokens. The t_1 transition is not enabled since p_1 has tokens and the t_1 transition would require p_1, p_2, and p_3 to have tokens.

In Figure 8, step 2, the t_2 transition has fired and a token is removed from p_3 and a token is added to p_1.

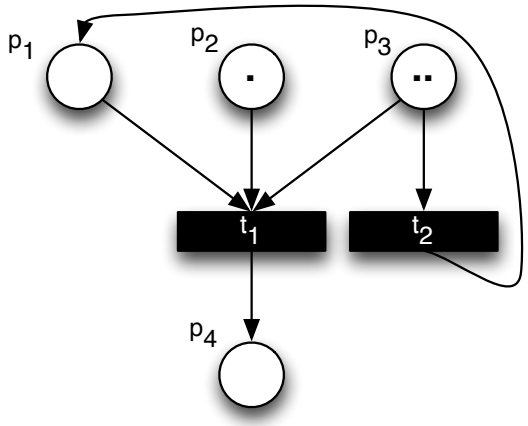

Figure 6: Marked Petri Net Visual Example

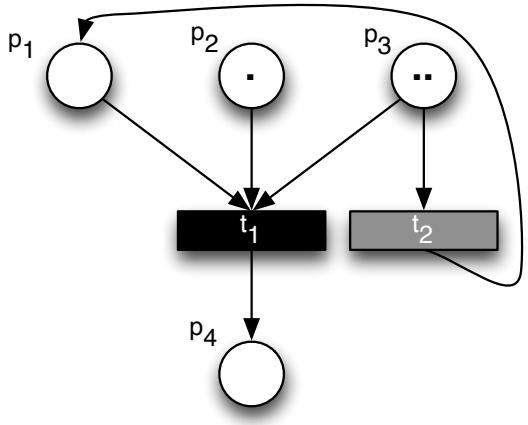

Figure 7: Marked Petri Net Visual Example, Executing Step 1

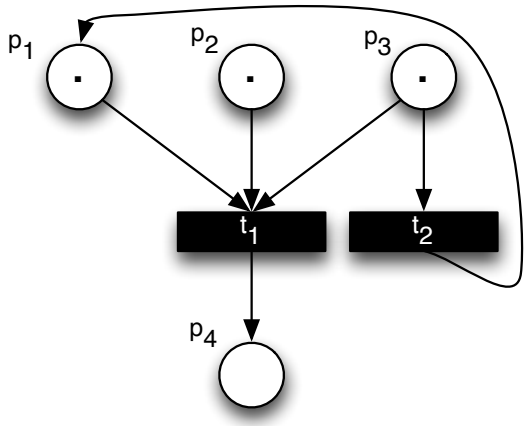

Figure 8: Marked Petri Net Visual Example, Executing Step 2

In Figure 9, step 3, both the t_1 and t_2 transitions are enabled, as all the places connected by inputs to the transitions have tokens. However, only one of the transitions can fire at a time, which will leave too few tokens remaining for the other transition to fire.

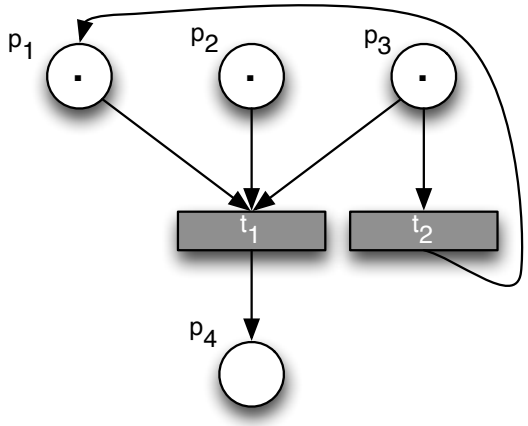

Figure 9: Marked Petri Net Visual Example, Executing Step 3

In Figure 10, step 4, we show the result of transition t_1 firing, which removes tokens from p_1, p_2, and p_3 while adding a token to p_4. In this case we selected t_1 to fire, despite both t_1 and t_2 being active based on their inputs.

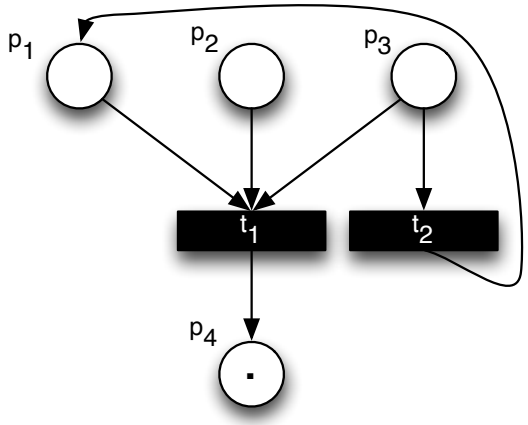

Figure 10: Marked Petri Net Visual Example, Executing Step 4

Finally, there are no more transitions active in the marked Petri net and execution has completed.

2.3 Inhibitor Arc Marked Petri Net

It has been noted that marked Petri nets are limited in their representative capabilities and are "too simple and limited to easily model real systems."[16, p. 190] Several extensions have been proposed which affectively increase the representative capability of Petri nets. The simplest extension is inhibitor arcs, where an input place to a transition can be required to have zero tokens for the transition to be enabled. All other extensions of Petri nets are no more than equivalent, from the perspective of accepting specific classes of formal languages, to Petri nets with inhibitor arcs.[16, p. 195-196] This extension is defined in Definition 3.

Definition 3. *A marked Petri net with* inhibitor arcs *is a 6-tuple* $(P,T,I,I_{inhibit},O,M)$ *in which* (P,T,I,O,M) *is a marked Petri net and* $I_{inhibit}$ *is a set of edges where* $I_{inhibit} \subseteq P \times T$.

In Figure 11 there is an added inhibitor arc represented by a line connecting a place (p_2) to a transition (t_2) with a circle at the transition end. This means that for transition t_2 to be enabled, there must be at least one token in p_3 and zero tokens in p_2.

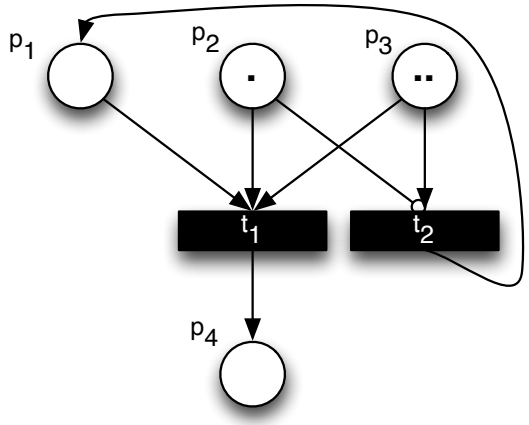

Figure 11: Marked Petri Net with Inhibitor Arcs Visual Example

This Petri net with inhibitor arcs (Figure 11) can also be described mathematically using Definition 3 as a Petri net, C, where:

$$
\begin{aligned}
C &= (P,T,I,I_{inhibit},O,M) \\
P &= \{p_1,p_2,p_3,p_4\} \\
T &= \{t_1,t_2\} \\
I(t_1) &= \{p_1,p_2,p_3\} \\
I(t_2) &= \{p_3\} \\
I_{inhibit}(t_1) &= \emptyset \\
I_{inhibit}(t_2) &= \{p_2\} \\
O(t_1) &= \{p_4\} \\
O(t_2) &= \{p_1\} \\
M &= (0,1,2,0)
\end{aligned}
$$

Execution is identical to that of a marked Petri net, with the exception of zero testing where inhibitor arcs exist. All other behavioral properties remain the same while the ability to recognize a greater set of classes of formal languages. In fact, a marked Petri net with inhibitor arcs for zero testing is equivalent to a Turing machine.[16, p. 201] This is an important distinction between marked Petri nets, which can only recognize languages that are no more expressive than context sensitive. In fact, there exist context free languages that marked Petri nets cannot recognize.[16, p. 184] This shows a more definitive reason for the added complexity of inhibitor arcs.

2.4 Inhibitor Arc Event Driven Marked Petri Net

It should be noted that the Petri nets that have been presented thus far are all closed systems. They do not naturally interact with other systems or the environment around them. Fortunately an extension has been created to address interaction from outside of the closed system: Event-Driven Petri nets, which are defined in Definition 4.

Definition 4. *A marked* Event-Driven *Petri net with inhibitor arcs is a 7-tuple* $(E_{in},E_{out},P,T,I,I_{inhibit},O,M)$ *in which* $(P,T,I,I_{inhibit},O,M)$ *is a marked Petri net with inhibitor arcs where:*

- E_{in} *is a set of input events,*
- E_{out} *is a set of output events,*
- I *and* $I_{inhibit}$ *map from* T *to* $P \cup E_{in}$*, and*
- O *maps from* T *to* $P \cup E_{out}$*. [9, p. 137]*

Events, both in and out, are treated identically to places except that they reference activities outside of the Petri net. A graphical example is shown in Figure 12, where e_1 and e_2 are input and output events, respectively.

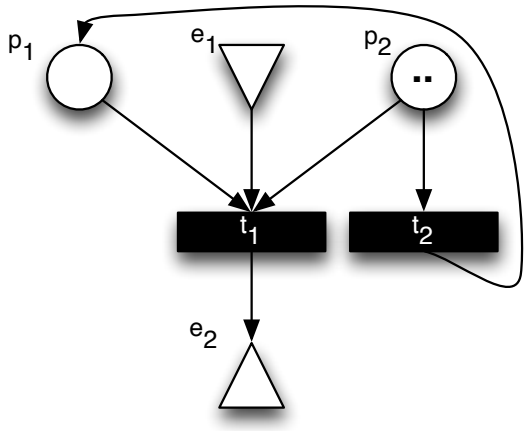

Figure 12: Marked Event Driven Petri Net with Inhibitor Arcs Visual Example

3 Statecharts

Statecharts were developed initially by David Harel as "a broad extension of the conventional formalism of state machines and state diagrams."[4] This variant of state machines, called Harel Statecharts, used the concept of Venn diagrams and directed graphs to express hierarchy and connectedness in a way that sought to solve the problem of *state explosion* present in typical state machines.[9, p. 155] This formalism has been co-opted into UML Statechart Diagrams, which now has its own formalisms and while similar to Harel Statecharts is definitely a variant of Harel Statecharts,[11] and is administered by the UML standards body.

UML is the standard for object modeling and contains diagrams for modeling a variety of components including structural, behavioral, and interaction.[9, p. 155] Specifically related to the work here, Statechart Diagrams are significantly more common than most any other forms of behavior modeling. This is due both to the ease of representation due to a rich array of descriptive features as well as inclusion in the UML standard and subsequently a strong set of tool support.

3.1 Types of Statecharts

Clearly there are two related types of Statecharts, both those defined by UML and the original proposed by David Harel. While the more commonly used are UML Statechart Diagrams, it is still useful to compare the two as a basis for the choice to use UML Statechart Diagrams.

Both Harel Statecharts and UML Statechart Diagrams are executed very similarly, although they are represented slightly differently. Transitions move execution between states. These transitions can be based on an event, guard, or always actionable (empty). When an event occurs, if the current state has a transition away from it that uses that event, the transition is taken. If a guard, which is a boolean expression, is evaluated to true at any point then that transition is taken. It is also possible to have transitions away from groups of states, which are denoted by a parent state away transition. Similarly transitions can go to a set of states, denoted by a transition to a parent state where the default sub-state is then transitioned to.

3.1.1 Harel Statecharts

Harel Statecharts were initially described in an informal way, using illustrative diagrams and describing the behavior of transitions.[4] However, more clarification and a formalism was necessary and was published shortly after the paper introducing Harel Statecharts.[5] Updates and new ways of formalizing Harel Statecharts are often defined when a new simulation or verification engine or methodology is created, not always staying entirely inline with the original.[19][13]

Harel Statecharts are made up of states defined as "blobs", where there is no strict hierarchical requirement imposed to allow for the blobs to be used as a Venn diagram that shows blobs containing portions of other blobs.[9, p. 155-156] An example that shows this lack of strict blob subsets is in Figure 13. You will see that state D is in both B and C. This would not be possible in UML Statechart Diagrams.

Definition 5. *A* Harel Statechart *is a 7-tuple* (s, S, t, p, e, d, E) *where,*

- S *is a finite set of states*
- s *is the start state and* $s \in S$

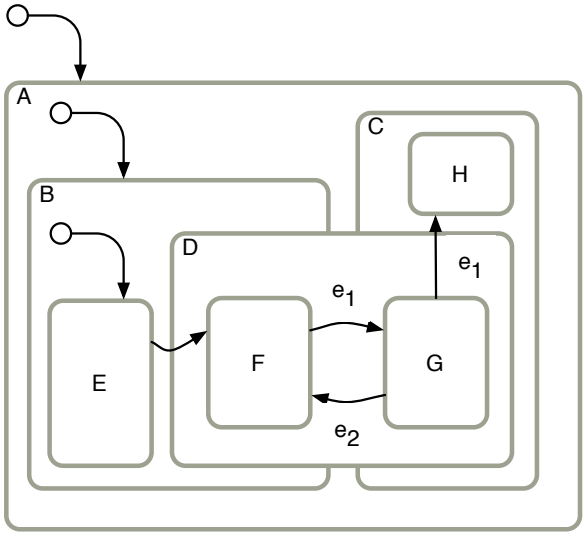

Figure 13: Harel Statechart Example

- *t is a transition function that maps the set of states, S onto a set were any element is in S*
- *p is a parent function that maps the set of states, S onto a set were any element is in S*
- *e is a event function that maps the set of transitions, $S \times S$ to a set of events.*
- *E is a finite set of events*
- *d is a default transition function that maps the the set of states to their default substate if it exists, or itself*

The Harel Statechart in Figure 13 can also be described mathematically using Definition 5 as a Harel Statechart, C, where:

$$
\begin{aligned}
C &= (s,S,t,p,e,d,E) \\
s &= A \\
S &= (A,B,C,D,E,F,G,H) \\
t &= (A,B,C,D,E,F,G,H) \rightarrow (\emptyset,\emptyset,\emptyset,\emptyset,(F),(G),(F,H),\emptyset) \\
p &= (A,B,C,D,E,F,G,H) \rightarrow (\emptyset,(A),(A),(A,B,C),(B),(D),(D),(C))\emptyset,\emptyset,\emptyset,\emptyset) \\
d &= (A,B,C,D,E,F,G,H) \rightarrow (B,E,C,D,E,F,G,H) \\
E &= (e_1,e_2,e_3)
\end{aligned}
$$

The event function e is:

	A	B	C	D	E	F	G	H
A	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
B	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
C	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
D	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
E	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
F	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	e_2	$\emptyset$
G	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	e_1	$\emptyset$	$\emptyset$
H	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	e_3	$\emptyset$

3.1.2 UML Statechart Diagrams

UML Statechart Diagrams are a variation of Harel Statecharts.[8] One of the largest differences between Harel Statecharts and UML Statechart Diagrams is that sub-states must be entirely contained within their parent. The remainder of the functionality, much of which is also included in Harel Statecharts, is broken into three categories.

Category I UML Statechart Diagrams are the simplest variety, and use simple states and transitions as well as initial states. *Category II* UML Statechart Diagrams add composite states, which subsequently show the need for entry transitions, exit transitions, completion transitions and final states. *Category III* UML Statechart Diagrams add the remaining Statechart features including guards, deferred events, activities, non-signal events, and actions that involve variables.[8] Just as work on formalizing Harel statecharts continued, the work formalizing UML Statechart Diagrams also continued.[17][20]

A Category I UML Statechart Diagram definition is shown in Definition 6, which is different from a Harel Statechart as all states are fully enclosed in a single parent state.

Definition 6. *A* Harel Statechart *is a 7-tuple* (s,S,t,p,e,d,E) *where,*

- S *is a finite set of states*
- s *is the start state and* $s \in S$
- t *is a transition function that maps the set of states,* S *onto a set were any element is in* S
- p *is a parent function that maps the set of states onto itself*
- e *is a event function that maps the set of transitions,* $S \times S$ *to a set of events.*
- E *is a finite set of events*
- d *is a default transition function that maps the the set of states to their default substate if it exists, or itself*

A Category I UML Statechart Diagram example is shown in figure 14, which is visibily different from a Harel Statechart as all states are fully enclosed in a single parent state.

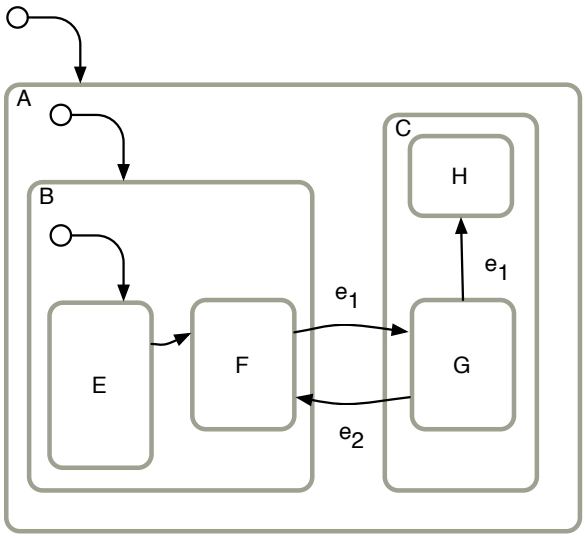

Figure 14: Category I UML Statechart Diagram Example

The Category I UML Statechart Diagram in Figure 14 can also be described mathematically using Definition 5 as a Harel Statechart, C, where:

$$
\begin{aligned}
C &= (s,S,t,p,e,d) \\
s &= A \\
S &= (A,B,C,E,F,G,H) \\
t &= (A,B,C,E,F,G,H) \rightarrow (\emptyset,\emptyset,\emptyset,\emptyset,(F),(G),(F,H),\emptyset) \\
p &= (A,B,C,E,F,G,H) \rightarrow (\emptyset,A,A,B,B,C,C)\emptyset,\emptyset,\emptyset,\emptyset) \\
d &= (A,B,C,E,F,G,H) \rightarrow (B,E,C,E,F,G,H) \\
E &= (e_1,e_2,e_3)
\end{aligned}
$$

The event function e is:

	A	B	C	E	F	G	H
A	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
B	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
C	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
E	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
F	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	e_2	$\emptyset$
G	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	e_1	$\emptyset$	$\emptyset$
H	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	e_3	$\emptyset$

Category II UML Statechart Diagrams can also be represented with Definition 6. The specific elements, such as entry and exit transtions can be represented by transitions to and from a parent state.

Category III UML Statechart Diagrams, however, require an additional set of components to allow for transitions to have guards and actions. This is shown in Definition 7 with the addtion of a, A, g, and G.

Definition 7. *A* UML Statechart Diagram Category III *is an 11-tuple* (s,S,t,p,e,d,E,a,A,g,G) *where,*

- *S is a finite set of states*
- *s is the start state and $s \in S$*
- *t is a transition function that maps the set of states, S onto a set were any element is in S*
- *p is a parent function that maps the set of states onto itself*
- *e is a event function that maps the set of transitions, $S \times S$ to a set of events.*
- *E is a finite set of events*
- *d is a default transition function that maps the the set of states to their default*
- *a is a action function that maps the set of transitions, $S \times S$ to a set of actions.*
- *A is a finite set of actions*
- *a is a guard function that maps the set of transitions, $S \times S$ to a set of guards.*
- *A is a finite set of guards substate if it exists, or itself*

4 Existing Translations

There has been a significant amount of previous work surrounding both UML Statechart Diagrams, Harel Statecharts, Petri nets of all varieties, and the SPIN model checker. In this section we attempt to focus in on specifically the areas that are complementary to the overall goal of this thesis, that

is, the translation of behavioral models that are easier to work with for modeling system behavior to behavioral models that are easier to analyze and verify with SPIN using PROMELA.

In 2001 Gannod and Gupta published details of an automated tool for analyzing Petri nets using SPIN.[3] Even more recently in 2008 in the area of distributed systems work has been done converting types of Petri nets for specific problems into PROMELA source code.[15]

Statecharts, both Harel[13][14] and UML Diagrams[10][18][12] have been studied intensively leading to translations directly from Statecharts to PROMELA.

Specific to the problems addressed in this thesis, there exists work on the conversions of Category I UML Statechart Diagrams to (colored) Petri nets [7], as well as Category II Statechart diagrams[8] by Hu and Shatz in 2004 and 2006, respectively. These methods were geared towards simulation as apposed to verification. However, these methods have recently been used for formal verification research.[2]

This gives the following translations that already exist in the research literature and are shown in Figure 15 in the approach section:

- Petri net to PROMELA,
- Statechart to PROMELA,
- Category I Statechart to Petri net,
- Category II Statechart to Petri net.

Part III

Approach

As seen in the background and related work the use of Statecharts, Petri nets, and PROMELA to model behavioral items and validate that model using SPIN is common, expected, and well studied. However, the translation of these behavioral models to a model checking formalism has two issues. First, the translations are not always clear and therefore not always traceable. It is important that when a validation is found, the issue is identifiable in the behavioral model. Similarly, it is important that it is clear what to validate once a behavioral model has been translated into a model checking formalism. Second, the entire breadth of UML Statechart Diagram features has not been translated to Petri nets. Specifically, guards and actions that involve variables are missing - yet they are an enormous set of functionality.

The approach is then two fold, re-create translations with a focus on traceability and add new translations for guards and actions that involve variables and expressions. This effectively fills in existing transitions while expanding them. Figure 15 shows a graphical representation of these translations.

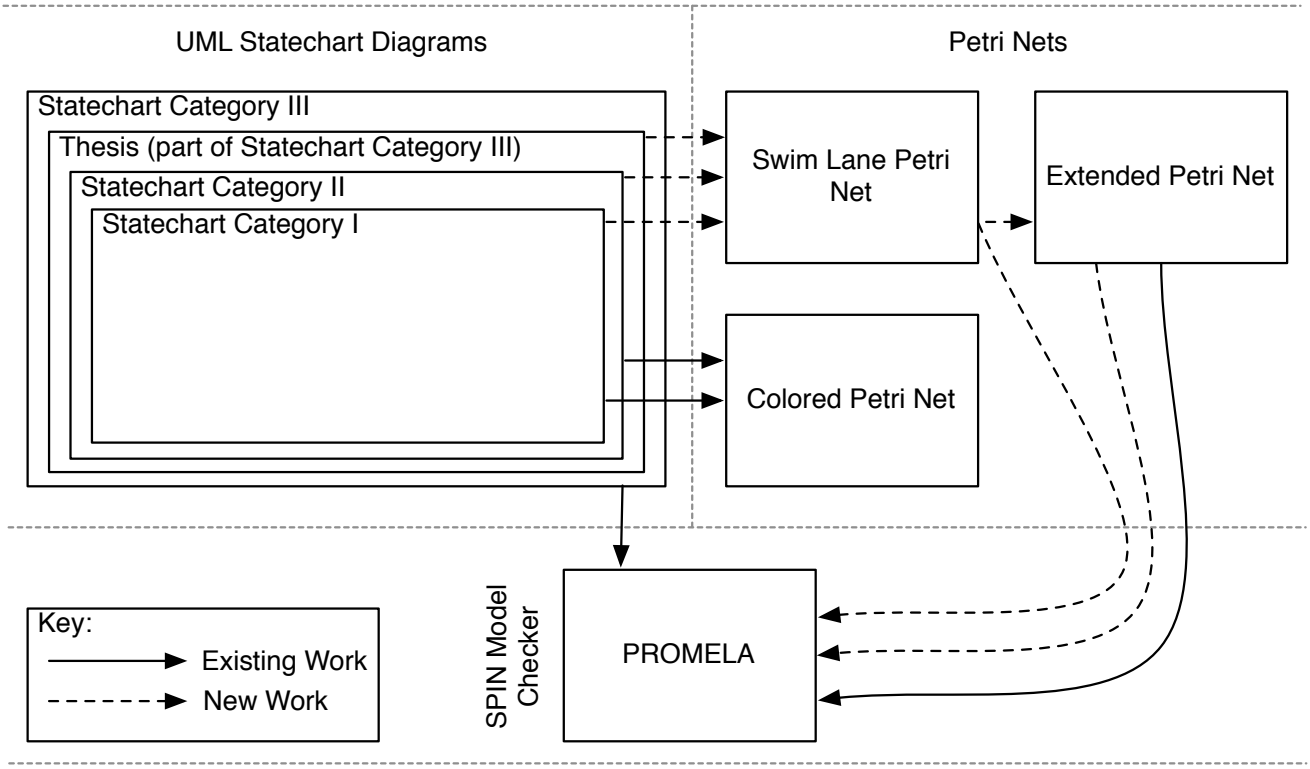

Figure 15: New and Existing Translations

5 Traceability

Traceability is a necessary goal for verification when a translation between representations is used. Certainly it is more difficult to find the origin of an issue in two unrelated but equivalent models than two models where there is some traceability between them. In an effort to ensure traceability, all translations are easily traced by the simple idea that a UML Statechart state becomes a Petri net place. This gives a type of coordinates between the models. Not only can states and places be easily mapped, but transitions between them are also found easily by finding the connections between two states or two places and identifying the translated pair.

6 Definitions

In addition to the models defined in the background section, we also make use of an extended notation for Petri nets that allows for concurrent regions, or swim lanes. This is a powerful concept,

as while Petri nets already simulate things in parallel, Swim Lane Petri nets concurrently simulate things in parallel, which can be very useful for systems of systems.

6.1 Swim Lane Marked Petri Net

Definition 8. *A* swim lane *marked Petri net is a 7-tuple* (P,T,I,O,M,L,N) *in which* (P,T,I,O,M) *is a marked Petri net and L is a set of n sets, where*

- $n \geq 1$ *is the number of swim lanes, and*
- $\bigcup_{i=1}^{n} L_i = P$, *and*
- $\bigcup_{i=1}^{n} N_i = T$.

An example of a swim lane marked Petri net can be found in Figure 16.

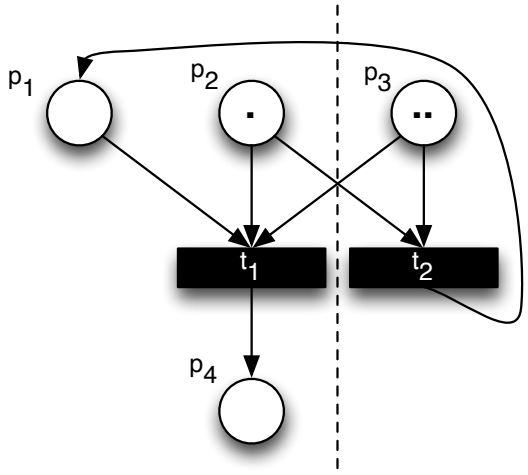

Figure 16: Example Swim Lane Marked Petri Net

This swim lane marked Petri net (Figure 16) can also be described mathematically using Definition 8 as a Petri net, C, where:

$$
\begin{aligned}
C &= (P,T,I,O,M,L,N) \\
P &= \{p_1,p_2,p_3,p_4\} \\
T &= \{t_1,t_2\} \\
I(t_1) &= \{p_1,p_2,p_3\} \\
I(t_2) &= \{p_2,p_3\} \\
O(t_1) &= \{p_4\} \\
O(t_2) &= \{p_1\} \\
M &= (0,1,2,0) \\
L &= \{\{p_1,p_2,p_4\},\{p_3\}\} \\
N &= \{\{t_1\},\{t_2\}\}
\end{aligned}
$$

6.2 Swim Lane Inhibitor Arc Marked Petri Net

Definition 9. *A* swim lane *inhibitor arc marked Petri net is a 8-tuple* $(P,T,I,I_{inhibit},O,M,L,N)$ *in which* $(P,T,I,I_{inhibit},O,M)$ *is an inhibitor arc marked Petri net and L is a set of n sets, where*

- $n \geq 1$ *is the number of swim lanes, and*
- $\bigcup_{i=1}^{n} L_i = P$, *and*
- $\bigcup_{i=1}^{n} N_i = T$.

An example of a swim lane marked Petri net can be found in Figure 17.

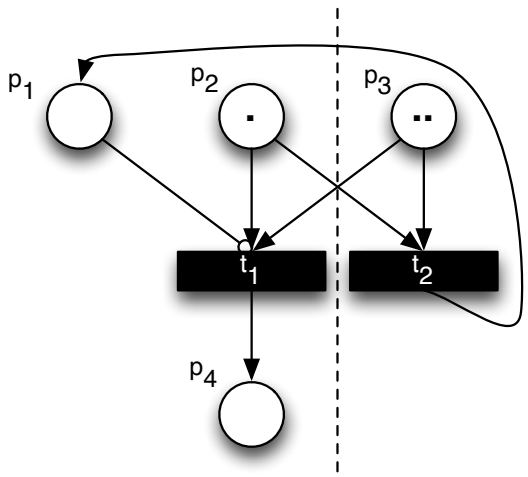

Figure 17: Example Swim Lane Inhibitor Arc Marked Petri Net

This swim lane marked Petri net (Figure 17) can also be described mathematically using Definition 9 as a Petri net, C, where:

$$
\begin{aligned}
C &= (P,T,I,I_{inhibit},O,M,L,N) \\
P &= \{p_1,p_2,p_3,p_4\} \\
T &= \{t_1,t_2\} \\
I(t_1) &= \{p_2,p_3\} \\
I(t_2) &= \{p_2,p_3\} \\
I_{inhibit}(t_1) &= \{p_1\} \\
I_{inhibit}(t_2) &= \{\emptyset\} \\
O(t_1) &= \{p_4\} \\
O(t_2) &= \{p_1\} \\
M &= (0,1,2,0) \\
L &= \{\{p_1,p_2,p_4\},\{p_3\}\} \\
N &= \{\{t_1\},\{t_2\}\}
\end{aligned}
$$

7 Algorithms

The algorithms in this section are, taken in groups or individually, the components which create the transitions shown in the approach section (Figure 15).

7.1 Inhibitor Arc Marked Petri net to PROMELA

The Algorithm listed in Algorithm 1 converts an inhibitor arc marked Petri net into PROMELA that can be simulated and verified with the SPIN model checker.

7.1.1 Algorithm Listing

Data: Inhibitor Arc Marked Petri net
Result: PROMELA
foreach *place* **do**
    create new global variable representing place;
    set new global variable equal to tokens or zero;
end
foreach *transition* **do**
    add logical expression representing inputs to transition;
    add action removing and adding tokens;
end
surround logical expressions with loop terminating upon no remaining transitions;

Algorithm 1: Inhibitor Arc Marked Petri net to PROMELA

7.1.2 Example

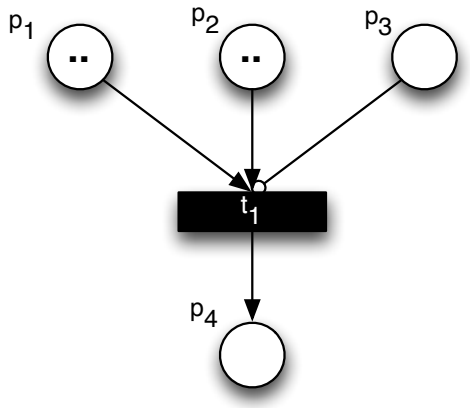

Figure 18: Example Inhibitor Arc Marked Petri Net to be Converted to PROMELA

The inhibitor arc marked Petri net in Figure 18 can be converted to PROMELA using Algorithm 1. For example, in the listing below we can see that there are four variables, each representing a place with the appropriate number of tokens. A loop executes the possible transitions, which are built up from the transition inputs, until no transitions are enabled and the *else* statement is executed.

```
int p_1 = 2;
int p_2 = 2;
```

```
int p_3 = 0;
int p_4 = 0;

active proctype petriNet() {
  do
  /* t_1 */
  :: p_1 > 0 && p_2 > 0 && p_3 == 0 -> p_1--; p_2--; p_4++;
  /* No transitions active */
  :: else -> break;
  od;
}
```

Adding multiple transitions is straightforward. In Figure 19, we add a transition t_2 and the PROMELA code is updated to include it.

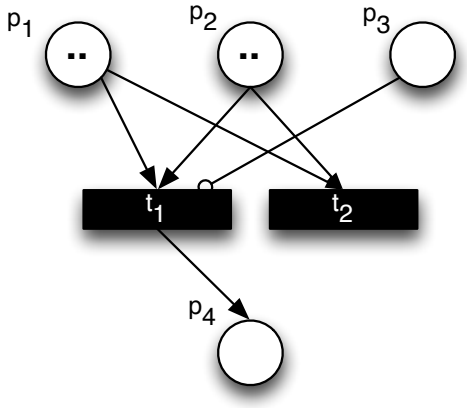

Figure 19: Example Inhibitor Arc Marked Petri Net with multiple transitions to be Converted to PROMELA

```
int p_1 = 2;
int p_2 = 2;
int p_3 = 0;
int p_4 = 0;

active proctype petriNet() {
  do
  /* t_1 */
  :: p_1 > 0 && p_2 > 0 && p_3 == 0 -> p_1--; p_2--; p_4++;
  /* t_2 */
  :: p_1 > 0 && p_2 > 0 -> p_1--; p_2--; p_4++;
  /* No transitions active */
  :: else -> break;
```

```
    od ;
}
```

Both t_1 and t_2 could be enabled at the same time, and therefore both conditionals for t_1 and t_2 would be true. However, since in PROMELA code only one of the true conditionals is executed the behavior expected from a Petri net is maintained. The possible transition firing orders are one of the following:

$$t_1, t_1$$
$$t_1, t_2$$
$$t_2, t_1$$
$$t_2, t_2$$

7.2 Swim Lane Inhibitor Arc Marked Petri Net to PROMELA

The Algorithm listed in Algorithm 2 converts a swim lane inhibitor arc marked Petri net into PROMELA that can be simulated and verified with the SPIN model checker.

7.2.1 Algorithm Listing

```
Data: Swim Lane Inhibitor Arc Marked Petri net
Result: PROMELA
foreach place do
    create new global variable representing place;
    set new global variable equal to tokens or zero;
end
foreach Swim Lane do
    add flag variable indicating if swim lane is stuck and set to unstuck;
    foreach transition do
        add logical expression representing inputs to transition;
        add action removing and adding tokens;
        set all swim lane flags to unstuck;
    end
    surround logical expressions with loop terminating if all swim lanes are stuck;
    add condition for no currently remaining transitions to set swim lane to stuck;
end
```

Algorithm 2: Swim Lane Inhibitor Arc Marked Petri Net to PROMELA

7.2.2 Example

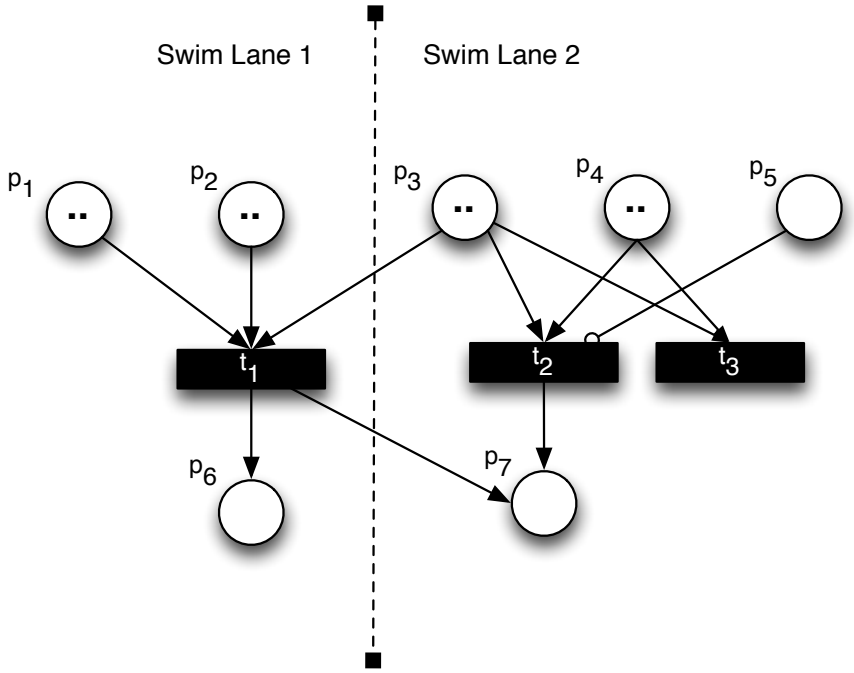

Figure 20: Example Swim Lane Inhibitor Arc Marked Petri Net to be Converted to PROMELA

The swim lane inhibitor arc marked Petri net in Figure 20 can be converted to PROMELA using Algorithm 2. The method is similar to the inhibitor arc marked Petri net method, with additional checks for when there are no further transitions, and a concurrent process for each swim lane.

```
int p_1 = 2;
int p_2 = 2;
int p_3 = 2;
int p_4 = 2;
int p_5 = 0;
int p_6 = 0;
int p_7 = 0;
bool SL1_Stuck = false;
bool SL2_Stuck = false;

active proctype petriNetSL1() {
  do
  :: true -> atomic {
    if
    /* t_1: */
    :: p_1 > 0 && p_2 > 0 && p_3 > 0 ->
      p_1--; p_2--; p_3--; p_6++; p_7++;
      SL1_Stuck = false; SL2_Stuck = false;
    /* If all Swim Lanes stuck, exit */
    :: SL1_Stuck == true && SL2_Stuck == true -> break;
    /* No transitions fired */
    :: else -> SL1_Stuck = true;
    fi
  };
  od;
}
active proctype petriNetSL2() {
  do
  :: true -> atomic {
    if
    /* t_2: */
    :: p_3 > 0 && p_4 > 0 && p_5 == 0 -> p_3--; p_4--; p_7++;
      SL1_Stuck = false; SL2_Stuck = false;
    /* t_3: */
    :: p_3 > 0 && p_4 > 0 -> p_3--; p_4--; p_7++;
      SL1_Stuck = false; SL2_Stuck = false;

    /* If all Swim Lanes stuck, exit */
    :: SL1_Stuck == true && SL2_Stuck == true -> break;
```

```
    /* No transitions fired */
    :: else -> SL2_Stuck = true;
    fi
  };
  od;
}
```

7.3 UML Statechart Category I to Event Driven Petri Net

Algorithm 3 converts a Category I UML Statechart to an inhibitor arc event driven marked Petri net.

7.3.1 Algorithm Listing

```
Data: UML Statechart Category I
Result: Inhibitor Arc Event Driven Makred Petri Nets
foreach state and substate do
    create petri net place;
end
foreach transition do
    add petri net transition;
    if transition has an input event then
        if input event does not exist then
            create input event;
        end
        add input event linked to the transition input;
    end
    if transition has an output event then
        if output event does not exist then
            create output event;
        end
        add output event linked to the transition output;
    end
    connect UML Statechart transition output to petri net transition input;
    connect UML Statechart transition input to petri net transition output;
end
foreach default sub-state do
    add petri net transition;
    connect UML Statechart state to Petri net transition input;
    connect petri net transition output to Petri net place representing UML Statechart default
    sub-state;
end
set token count to 1 in Petri net place representing top level initial state;
```

Algorithm 3: UML Statechart Category I to Inhibitor Arc Event Driven Marked Petri Net

7.3.2 Example

In Figure 22, which is converted from Figure 21 using Algorithm 3, there are three input events, as well as 7 total transitions, one for each of the transitions in the original Statechart and one transition for the default sub-state in state *A*. Place *A* also has a token as it is the initial state of the entire Statechart.

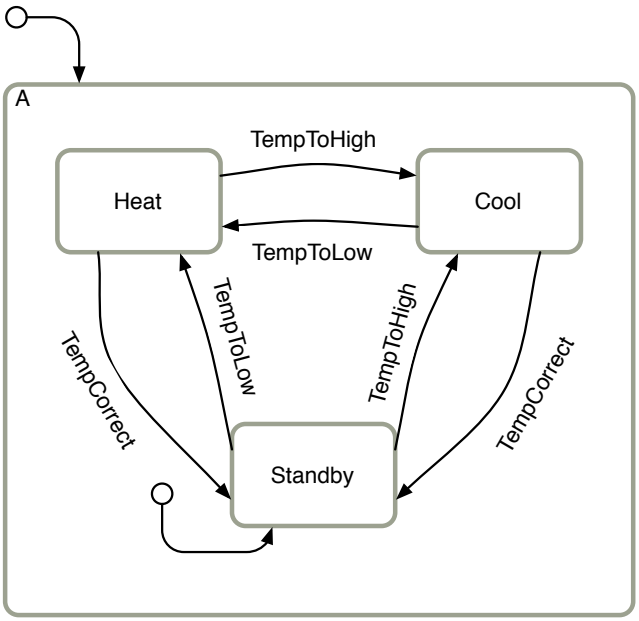

Figure 21: Example Category I UML Statechart Diagram to be converted to an Inhibitor Arc Event Driven Marked Petri Net

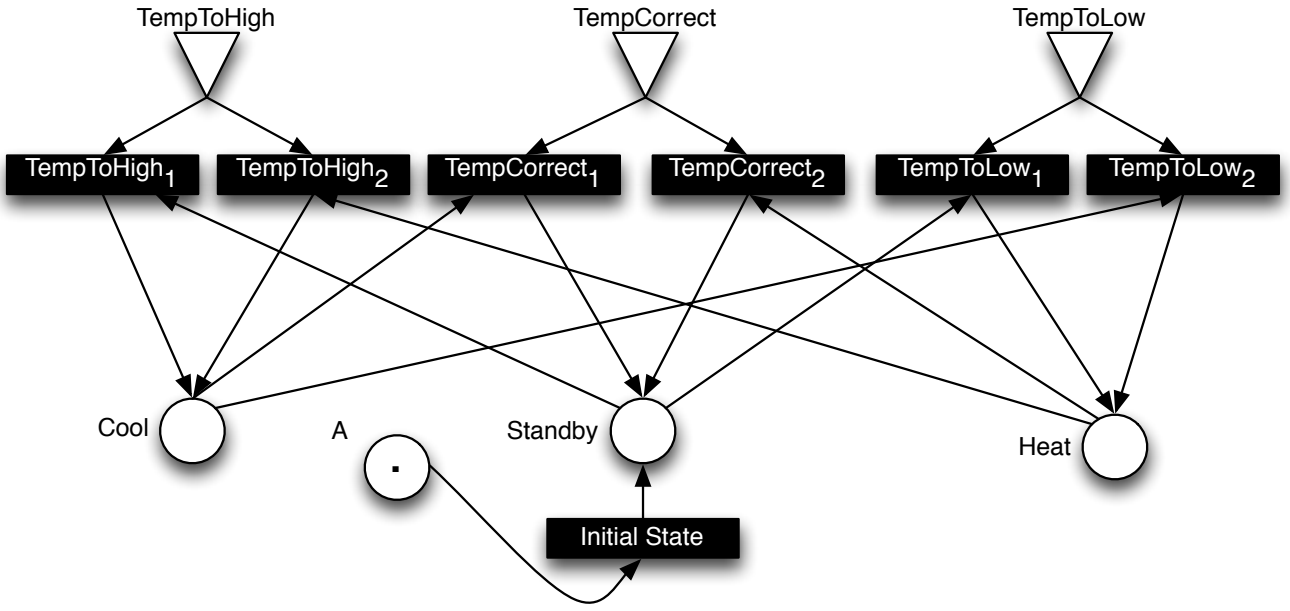

Figure 22: Inhibitor Arc Event Driven Marked Petri Net converted from Figure 20 by Algorithm 3

7.4 Removal of Entry Transitions from UML Statechart Category II

Entry transitions are transitions that go to a parent state indicating the same transition exists to every child state. The algorithm removes the entry transition and creates additional transitions for every child state. If the child state also has sub-states, the process is repeated until no further entry transitions exist.

7.4.1 Algorithm Listing

```
Data: UML Statechart Category II with Entry Transitions
Result: UML Statechart Category II without Entry Transitions
while there exist entry transition to state with substates exists do
    move transition from state to default substate;
end
foreach default entry do
    if not top level initial state then
        remove default entry;
    end
end
```

Algorithm 4: Removal of Entry Transitions from UML Statechart Category II

7.4.2 Example

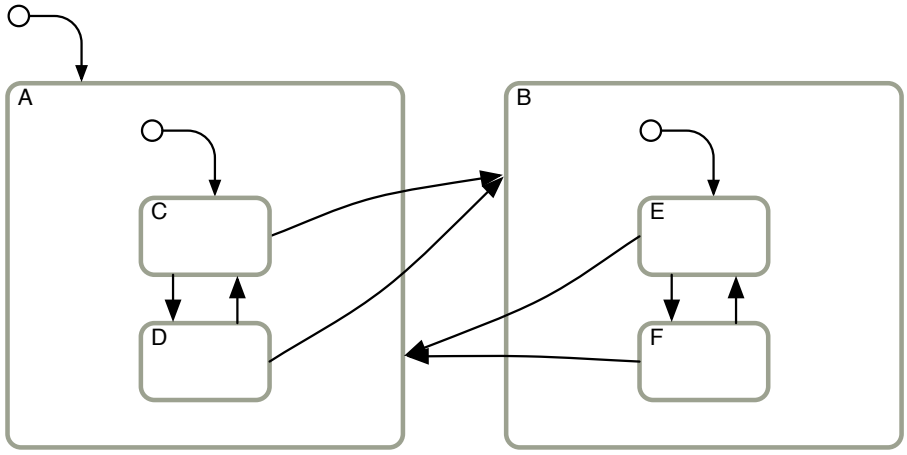

Figure 23: Example Category II Statechart Diagram with Entry Transitions

The UML Statechart Diagram with entry transitions in Figure 23 is converted by Algorithm 4 into a UML Statechart Diagram that has no entry transitions in Figure 24. The transitions in Figure

23 that go from both *C* and *D* to *B* are moved so that they now go to *E* and the default transition into *E* is removed in Figure 24. Similarly the transitions that go from *E* and *F* to *A* in Figure 23 are moved so that they both go to *C* in Figure 24 and the default transition into *C* is removed.

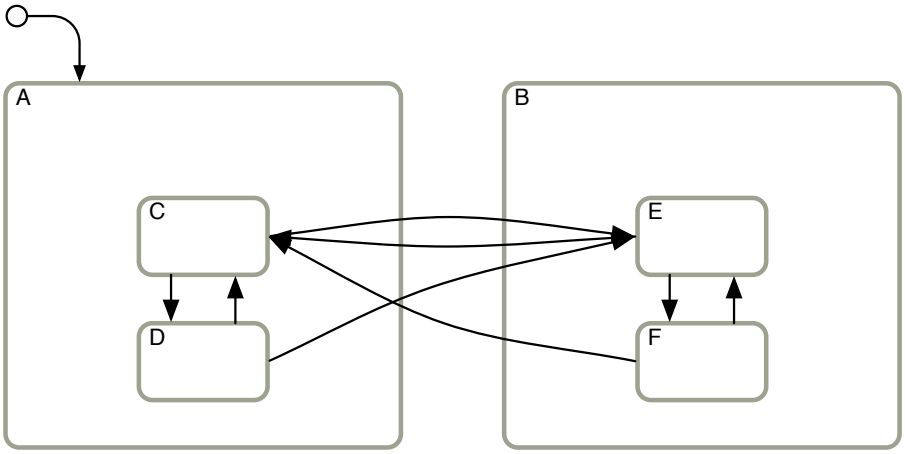

Figure 24: Example Category II Statechart Diagram with Entry Transitions Removed

7.5 Removal of Exit Transitions from UML Statechart Category II

Exit transitions are transitions that leave a parent state indicating the same transition exists from every child state. The algorithm removes the exit transition and creates additional transitions for every child state. If the child state also has sub-states, the process is repeated until no further exit transitions exist.

7.5.1 Algorithm Listing

```
Data: UML Statechart Category II with Exit Transitions
Result: UML Statechart Category II without Exit Transitions
while exit transtion from state with substates exist do
    select state;
    foreach substate do
        add transition to state exit transition destination from current substate
    end
    remove original exit transtion;
end
```

Algorithm 5: Removal of Exit Transitions from UML Statechart Category II

7.5.2 Example

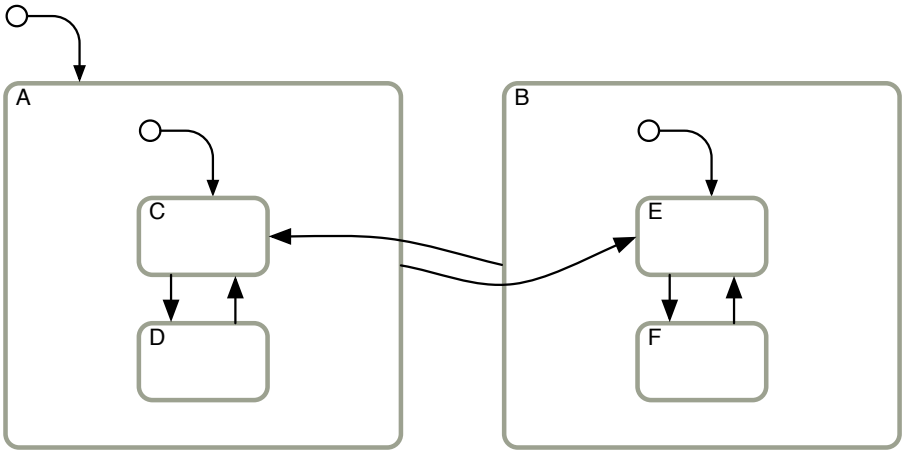

Figure 25: Example Category II Statechart Diagram with Exit Transitions

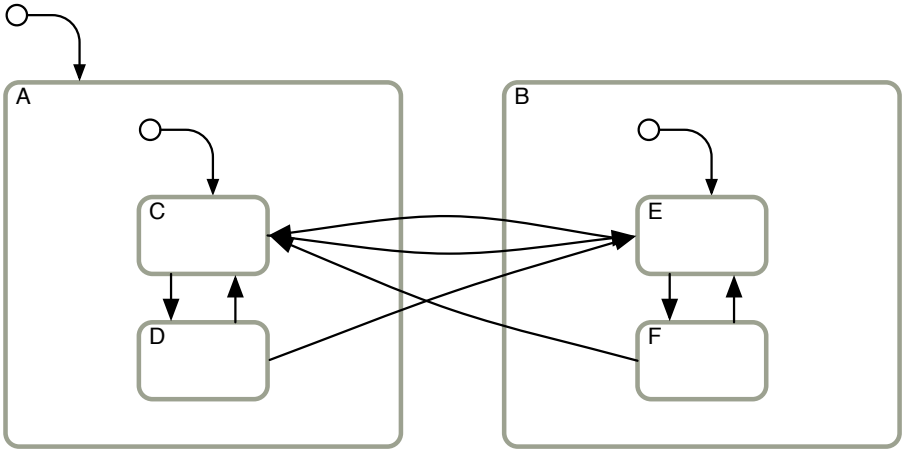

Figure 26: Example Category II Statechart Diagram with Exit Transitions Removed

7.6 Removal of Completion Transitions from UML Statechart Category II

Completion transition is a special type of exit transition where the state that is being left is orthogonal and therefore has multiple concurrent regions. The completion transition joins these multiple concurrent paths of execution into one.

7.6.1 Algorithm Listing

```
Data: UML Statechart Category II with Completion Transitions
Result: UML Statechart Category II without Completion Transitions
while completion transitions exist do
    if contains orthogonal regions then
        add transition to completion transition with join from each orthogonal region;
        foreach orthogonal region do
            if contains substates then
                add new state;
                foreach state with no out transitions within current orthogonal region do
                    add new transition from current state to new state;
                end
                remove completion transition;
                add transition from new state to forked transition input;
            end
        end
    end
end
```

Algorithm 6: Removal of Completion Transitions from UML Statechart Category II

7.6.2 Example

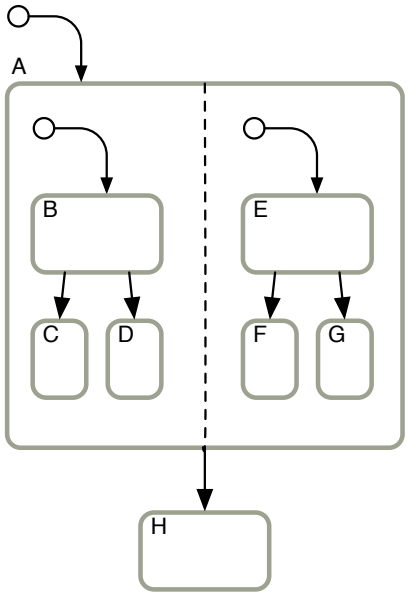

Figure 27: Example Category II Statechart Diagram with a Completion Transition

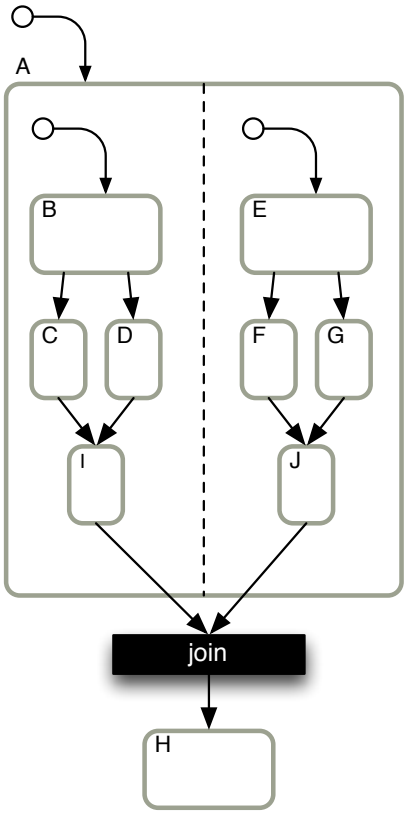

Figure 28: Example Category II Statechart Diagram with a Completion Transition Removed

7.7 Swim Lane Inhibitor Arc Marked Petri Net to Inhibitor Arc Marked Petri Net

Conversion from swim lane inhibitor arc marked Petri net to inhibitor arc marked Petri nets is trivial in the case where there is no contention by transitions for tokens across swim lanes. However, it may not always be the case that there is no contention between swim lanes. In that case, despite the fact that each swim lane is operating in parallel it is necessary to allow for one swim lane's transition to fire first and deprive the other potential transitions of the tokens they needed. Given these restrictions, Algorithm 7 converts from swim lane inhibitor arc marked Petri net to inhibitor arc marked Petri net.

7.7.1 Algorithm Listing

```
Data: Swim Lane Inhibitor Arc Marked Petri Net
Result: Inhibitor Arc Marked Petri Net
foreach swim lane do
    foreach place do
        move place to the first swim lane;
    end
    foreach transition do
        move transition to the first swim lane;
    end
    if swim lane is not the first swim lane then
        remove swim lane;
    end
end
```

Algorithm 7: Swim Lane Inhibitor Arc Marked Petri Net to Inhibitor Arc Marked Petri Net

7.7.2 Example

Functionally this has removed the swim lanes allowing the parallel regions to share one execution space. None of the transitions or arcs has been changed, outside of their swim lane location.

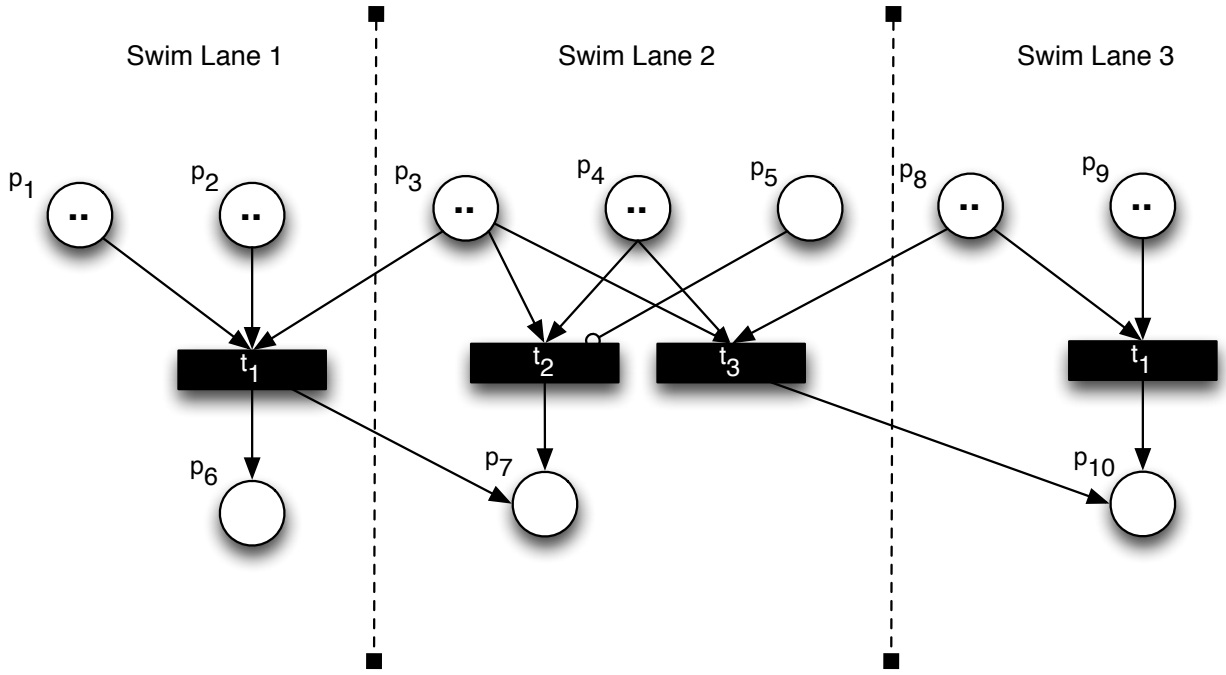

Figure 29: Swim Lane Inhibitor Arc Marked Petri Net

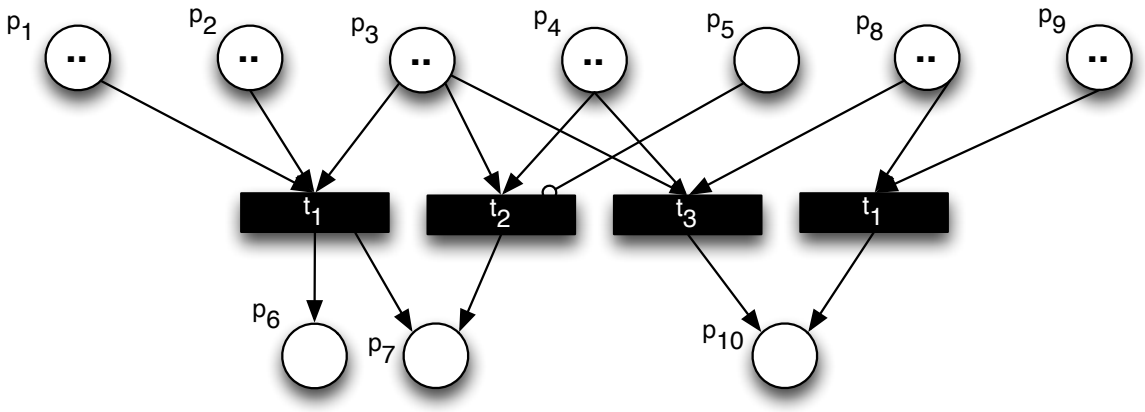

Figure 30: Inhibitor Arc Marked Petri Net created from Figure 29 using Algorithm 7

7.8 UML Statechart Category II with only Orthogonal Regions to Swim Lane Inhibitor Arc Marked Event Driven Petri Net

A major transition is from more complicated Statecharts, specifically those with concurrent regions, to swim lane inhibitor arc marked event driven Petri nets. In this transition swim lanes are used for the Petri nets due to the reduced complexity of transitioning regions that show clear parallel execution paths into a non-swim lane Petri net.

7.8.1 Algorithm Listing

The Algorithm is broken into two parts, first the portion that creates the Petri net places from the Statechart (Algorithm 8), then the addition of Petri net transitions and arcs to represent the Statechart transitions, forks, joins, default and initial states (Algorithm 9).

```
Data: UML Statechart Category II with only Orthogonal Regions
Result: Petri Net Places
create a default swim lane;
foreach state and substate do
    if state is orthogonal then
        foreach parallel region do
            create a swim lane;
        end
    end
    create petri net place;
    if parent state is orthogonal then
        place petri net place in swim lane corresponding to parallel region in parent state;
    end
    else if parent is in non-default swim lane then
        place petri net place in parent's swim lane;
    end
    else
        place petri net place in default swim lane;
    end
end
```

Algorithm 8: UML Statechart Category II with only Orthogonal Regions to Swim Lane Inhibitor Arc Marked Event Driven Petri Net, Part I

```
Data: UML Statechart Category II with only Orthogonal Regions
Result: Petri Net Transitions and Arcs
foreach transition do
    if transition is to a fork or join AND fork or join transition already exists then
        use existing petri net transition;
    end
    else
        add petri net transition;
    end
    if transition has an input event then
        if input event does not exist then
            create input event;
        end
        add input event linked to the transition input;
    end
    if transition has an output event then
        if output event does not exist then
            create output event;
        end
        add output event linked to the transition output;
    end
    if transition not used as fork that already existed then
        connect UML Statechart transition output to petri net transition input;
    end
    if transition not used as join that already existed then
        connect UML Statechart transition input to petri net transition output;
    end
end
foreach state with default sub-states do
    add petri net transition and place in swim lane of state;
    connect UML Statechart state to Petri net transition input;
    connect petri net transition output to all Petri net places representing UML Statechart
    default sub-state;
end
set token count to 1 in Petri net place representing top level initial state;
```

Algorithm 9: UML Statechart Category II with only Orthogonal Regions to Swim Lane Inhibitor Arc Marked Event Driven Petri Net, Part II

7.8.2 Example

In Figure 31 there is a UML Statechart Diagram with orthogonal states, sub-states, a join, default and inital states, and events. Each of these items must be explicitly handled in the conversion to a Petri net.

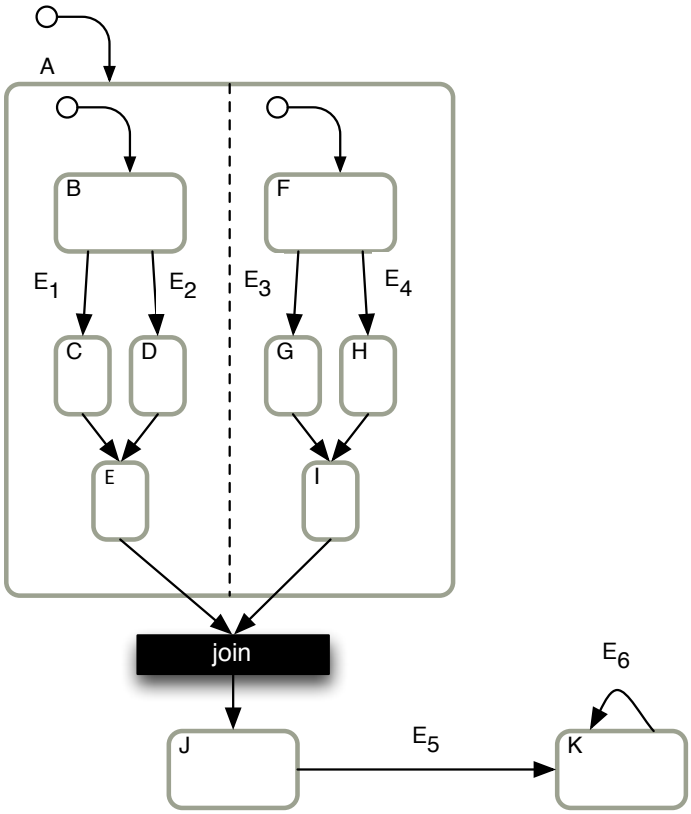

Figure 31: Example Category II Statechart Diagram with Only Orthogonal Regions

The individual elements in Figure 31 are handled as such:

- Orthogonal states create additional parallel regions by creating swim lanes for each additional parallel region.
- Sub-states are uniquely identified and broken out into places, as well as the parent states.
- Joins are handled by ensuring the appropriate Petri net arcs both go into a single Petri net transition.
- Default and initial states are handled by additional transitions and adding a token, respectively.

- Input events are added to the appropriate Petri net transitions to ensure they are only enabled upon an event.

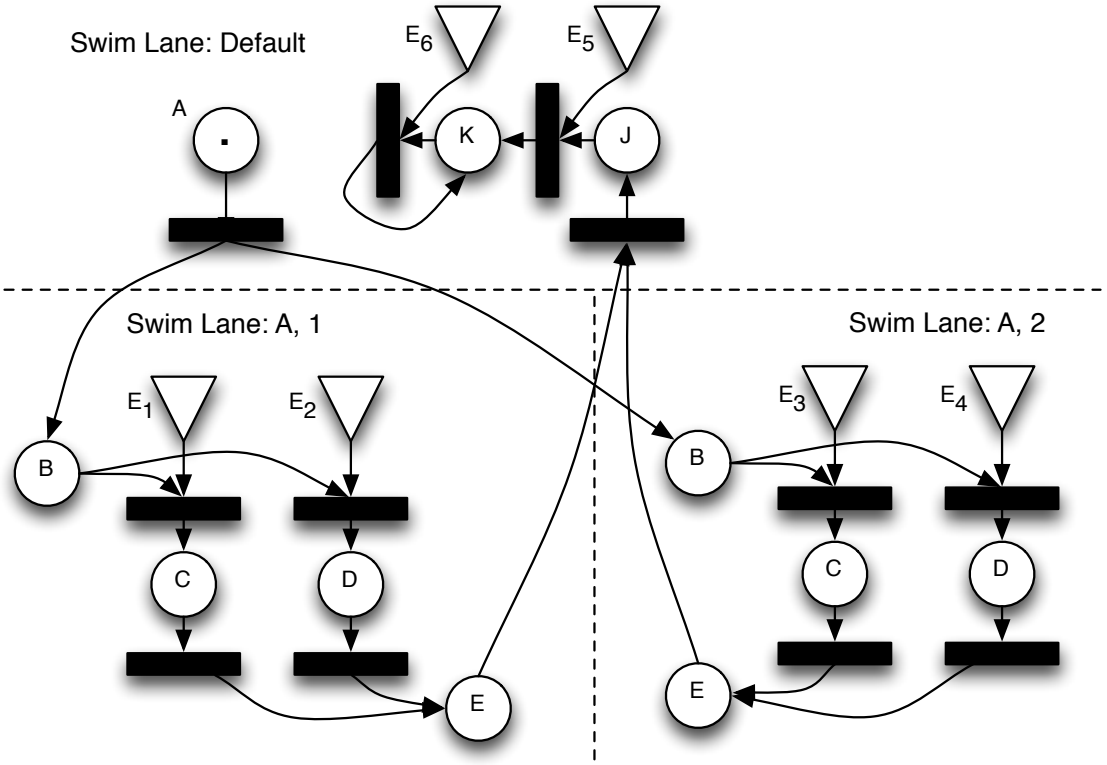

Figure 32: Example Category II Statechart Diagram with Only Orthogonal Regions converted to Petri Net

7.9 UML Statechart Category III Expressions with Variables to Swim Lane Inhibitor Arc Marked Event Driven Petri Net

One of the items that is possibly least clearly translated from UML Statechart Diagrams to extended Petri nets is expressions, and specifically those that involve variables. Since extended Petri nets do not have any natural method of expressing expressions, but are computationally capable of calculating them, the translation is more involved. To that end, what is presented here is a novel framework for representing expressions in inhibitor arc marked Petri nets. There are several limitations, including only using integer numbers and limited operations. These are not intrinsic to the

framework presented, but rather to the example of the framework being used. It would be possible to define more operators and other data types within the framework.

Expressions in UML Statechart Diagrams are used in two scenarios, actions and guards. Actions are elements where expressions are executable. Guards return boolean values to determine if a transition can be taken. The framework presented here allows for both uses of expressions, depending on the return value of the expression.

The framework presumes an abstract syntax tree has been created from the expression, which is then used to create an inhibitor arc marked Petri net where:

- Variables are represented by two Petri net places, one for magnitude and one for sign.
- Operators act on these variables and are a set of Petri net places, transitions, and arcs that have both inputs and outputs. The inputs and output places are shared with the corresponding output and input places in the other operators, respective.
- A control Petri net that allows operators to execute when all previous operators have executed.
- Output places of operators are assumed to be empty except for assignment operators.

Figure 33 shows the general structure of the framework for the expression $|Var_3(Var_1 + Var_2)|$. The chain of places on the left only allow for one operator to execute at a time. The operators, which are represented as boxes containing Petri nets, have inhibitor arcs to them, which symbolize an inhibitor arc to every single transition in the box. The half shown places are places that are both internal and external to the box, that is the interface. The Petri net chain on the left resets itself as operators are executed, finally outputting a token to a place indicating completion.

In addition to the operators shown, it is necessary to have a *copy* operator for each input variable. In the case of Figure 33, that would be Var_1 and Var_2. This is added to the Petri net chain on the left and they are executed as normal operations. Their function is to copy the Statechart wide variables non-destructively. The other operations remove all tokens from the Petri net places, so it is necessary to add the copy step. It is now shown, however, due to space.

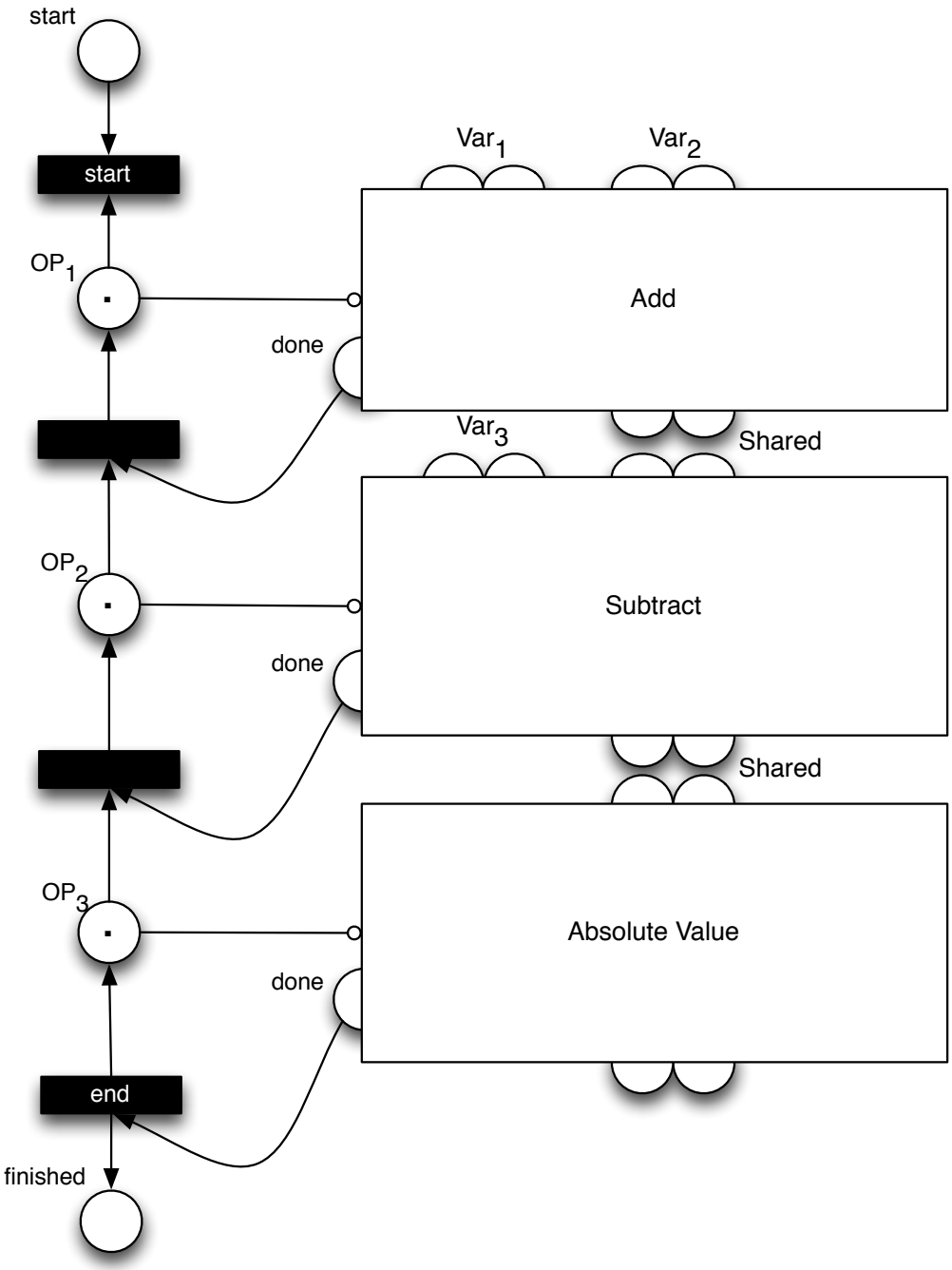

Figure 33: Petri Net Framework for UML Statechart Diagram Category III Expressions

As an example, Figure 34 shows an assignment operator. First, it removes all tokens from the output variables (in this case $output_{sign}$ and $output_{mag}$, then copies all tokens from the input variables ($input_{sign}$ and $input_{mag}$) to the output variables. Finally all internal places are reset to their original values and the *done* place is output with a token.

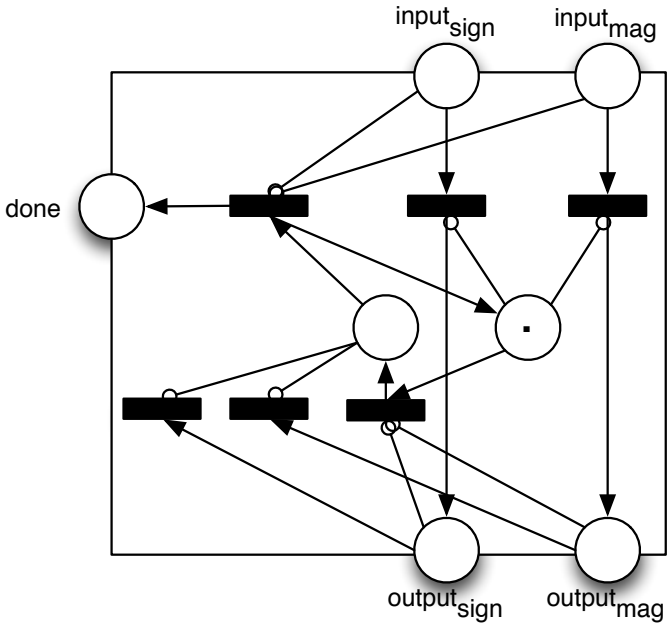

Figure 34: Petri Net Framework for UML Statechart Diagram Category III Expressions: Assignment Operator

The addition operator, which is complicated enough to require a diagram which exceeds the space available on a single page. As such, it is included in its mathematical form using definition 3 where C is an inhibitor arc marked Petri net.

$$
\begin{aligned}
C &= (P,T,I,I_{inhibit},O,M) \\
P &= \{p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8,p_9,p_{10},p_{11},p_{12},p_{13},p_{14},p_{15},p_{16},p_{17},p_{18}\} \\
T &= \{t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9,t_{10},t_{11},t_{12},t_{13},t_{14},t_{15},t_{16},t_{17},t_{18},t_{19},t_{20},t_{21},t_{22},t_{23},t_{24}\} \\
M &= (0,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0)
\end{aligned}
$$

The arcs to the transitions are defined as:

$$
\begin{aligned}
I(t_1) &= \{p_1, p_3\} \\
I(t_2) &= \{p_3\} \\
I(t_3) &= \{p_1\} \\
I(t_4) &= \{\emptyset\} \\
I(t_5) &= \{p_6, p_7\} \\
I(t_6) &= \{p_5\} \\
I(t_7) &= \{p_8\} \\
I(t_8) &= \{p_4\} \\
I(t_9) &= \{p_2\} \\
I(t_{10}) &= \{p_5\} \\
I(t_{11}) &= \{p_5, p_{16}\} \\
I(t_{12}) &= \{p_5\} \\
I(t_{13}) &= \{p_5, p_{16}\} \\
I(t_{14}) &= \{p_2, p_4\} \\
I(t_{15}) &= \{p_{11}, p_{13}, p_{14}\} \\
I(t_{16}) &= \{p_{10}, p_{12}, p_{14}\} \\
I(t_{17}) &= \{p_{15}\} \\
I(t_{18}) &= \{p_{15}\} \\
I(t_{19}) &= \{p_4, p_{16}\} \\
I(t_{20}) &= \{p_4\} \\
I(t_{21}) &= \{p_2\} \\
I(t_{22}) &= \{p_2, p_{16}\} \\
I(t_{23}) &= \{p_2\} \\
I(t_{24}) &= \{p_4\}
\end{aligned}
$$

The inhibitor arcs to the transitions are defined as:

$$
\begin{aligned}
I_{inhibit}(t_1) &= \{p_5\} \\
I_{inhibit}(t_2) &= \{p_1, p_5\} \\
I_{inhibit}(t_3) &= \{p_3, p_5\} \\
I_{inhibit}(t_4) &= \{p_1, p_3, p_9, p_5\} \\
I_{inhibit}(t_5) &= \{\emptyset\} \\
I_{inhibit}(t_6) &= \{p_1, p_2, p_3, p_4, p_7, p_8\} \\
I_{inhibit}(t_7) &= \{p_7\} \\
I_{inhibit}(t_8) &= \{p_7\} \\
I_{inhibit}(t_9) &= \{p_7\} \\
I_{inhibit}(t_{10}) &= \{p_1, p_2, p_3, p_4, p_{12}, p_{15}, p_{16}\} \\
I_{inhibit}(t_{11}) &= \{p_1, p_2, p_3, p_4, p_{12}, p_{15}\} \\
I_{inhibit}(t_{12}) &= \{p_1, p_2, p_3, p_4, p_{13}, p_{15}, p_{16}\} \\
I_{inhibit}(t_{13}) &= \{p_1, p_2, p_3, p_4, p_{13}, p_{15}\} \\
I_{inhibit}(t_{14}) &= \{p_{14}\} \\
I_{inhibit}(t_{15}) &= \{\emptyset\} \\
I_{inhibit}(t_{16}) &= \{\emptyset\} \\
I_{inhibit}(t_{17}) &= \{p_2, p_{14}\} \\
I_{inhibit}(t_{18}) &= \{p_4, p_{14}\} \\
I_{inhibit}(t_{19}) &= \{p_{12}, p_{15}\} \\
I_{inhibit}(t_{20}) &= \{p_{12}, p_{15}, p_{16}\} \\
I_{inhibit}(t_{21}) &= \{p_{12}, p_{15}\} \\
I_{inhibit}(t_{22}) &= \{p_{13}, p_{15}\} \\
I_{inhibit}(t_{23}) &= \{p_{13}, p_{15}, p_{16}\} \\
I_{inhibit}(t_{24}) &= \{p_{13}, p_{15}\}
\end{aligned}
$$

The arcs from the transitions to the places are defined as:

$$
\begin{aligned}
O(t_1) &= \{p_6, p_8\} \\
O(t_2) &= \{p_{10}\} \\
O(t_3) &= \{p_{11}\} \\
O(t_4) &= \{p_6, p_9\} \\
O(t_5) &= \{\emptyset\} \\
O(t_6) &= \{p_7, p_9\} \\
O(t_7) &= \{p_{17}\} \\
O(t_8) &= \{p_{18}\} \\
O(t_9) &= \{p_{18}\} \\
O(t_10) &= \{p_9, p_{12}, p_{15}, p_{16}\} \\
O(t_11) &= \{p_9, p_{12}, p_{15}, p_{16}\} \\
O(t_12) &= \{p_9, p_{13}, p_{15}, p_{16}\} \\
O(t_13) &= \{p_9, p_{13}, p_{15}, p_{16}\} \\
O(t_14) &= \{\emptyset\} \\
O(t_15) &= \{\emptyset\} \\
O(t_16) &= \{\emptyset\} \\
O(t_17) &= \{p_{14}\} \\
O(t_18) &= \{p_{14}\} \\
O(t_19) &= \{p_{17}, p_{18}\} \\
O(t_20) &= \{p_{18}\} \\
O(t_21) &= \{p_{18}\} \\
O(t_22) &= \{p_{17}, p_{18}\} \\
O(t_23) &= \{p_{18}\} \\
O(t_24) &= \{p_{18}\}
\end{aligned}
$$

The places represented by p_1 and p_2 represent the sign and magnitude of the first input, respectively, while p_3 and p_4 are the second input and the output is represented by p_{17} and p_{18}. The *done* place is represented by p_9. Using this newly defined addition operator, we can define a subtraction operators as well, as shown in figure 35. Note that this makes use of the addition operator within it, simply changing the sign of the second input. Note, in this figure, the done and input and output places are shared, except for Var_2's sign place which is altered to be the opposite of its current value.

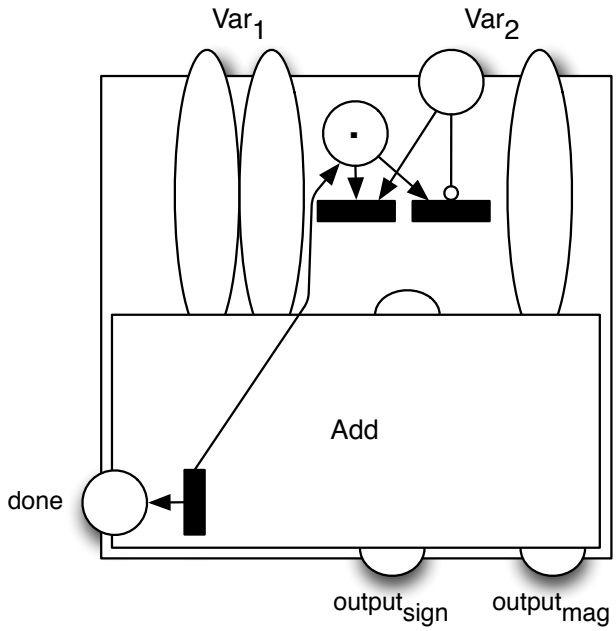

Figure 35: Petri Net Framework for UML Statechart Diagram Category III Expressions: Subtraction Operator

Simpler operators also exist, for example, the absolute value operator is very simple indeed and is represented in Figure 36. In this case all values are removed from the input variables and, for the magnitude, assigned to the output variable. Two important things must be noticed; first it is assumed that the output places are empty since it is not assignment. Second, we must take special care to ensure that the input sign is also empty even though we do not use its value.

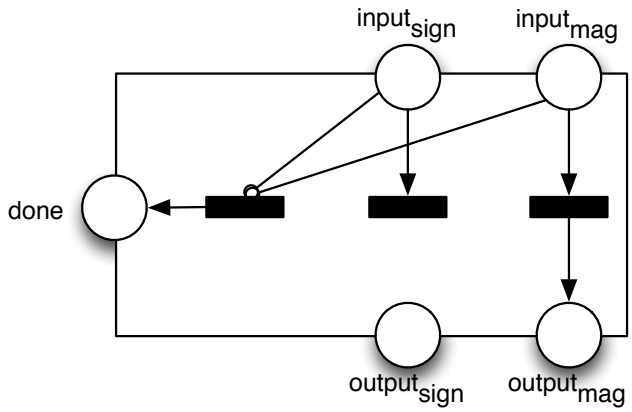

Figure 36: Petri Net Framework for UML Statechart Diagram Category III Expressions: Absolute Value Operator

7.9.1 Algorithm Listing

The Algorithm for this operation depends on the both the previous transformations as well as that the operators used are defined.

Data: UML Statechart Category III Expressions
Result: Inhibitor Arc Marked Petri Net
foreach *variable assigned* **do**
    create two Petri net places to represent the value;
end
translate to Inhibitor Arc Marked Petri Net ignoring guards and expressions;
foreach *guard* **do**
    replace the arc entering the transition that corresponds with the Statechart transition with the Petri Net framework liking the expression output place to the Petri net transition with an arc;
    add Petri net transition from framework done place;
    foreach *input variable* **do**
        add a copy operator to the Petri Net framework;
    end
end
foreach *action* **do**
    replace the arc entering the transition that corresponds with the Statechart transition with the Petri Net framework liking the framework done place to the Petri net transition with an arc;
    foreach *input variable* **do**
        add a copy operator to the Petri Net framework;
    end
end

Algorithm 10: UML Statechart Diagram Category III Expressions with Variables to Swim Lane Inhibitor Arc Marked Event Driven Petri Net

7.9.2 Example

In Figure 37 there is only two states with a transition that has a guard. First we assume this has been transformed into an inhibitor arc marked Petri net, then we apply Algorithm 10.

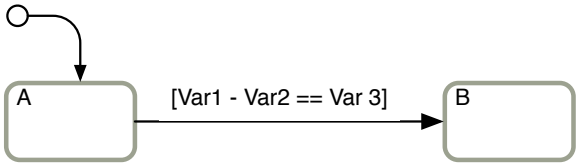

Figure 37: UML Category III Statechart Diagram with Guard

In Figure 38 state *A* has become a Petri net place, as has *B*. The transition between them has been modified to include the expression framework, which calls two different operators as the Petri net chain on the left side is executed. Petri net place *B*, which represents state *B* takes its input from a transition that is connected to the operators, while the indication that the Petri net chain is finished is sent into a destinationless transition.

Figure 39, which is very similar to Figure 37, there is only a simple action that requires execution during the transition.

In Figure 40 state *A* has become a Petri net place, as has *B*. The transition between them has been modified to include the expression framework, which calls two different operators as the Petri net chain on the left side is executed. Petri net place *B*, which represents state *B* takes its input from a transition that is connected to the Petri net chain finished place, the assignment operator has no output.

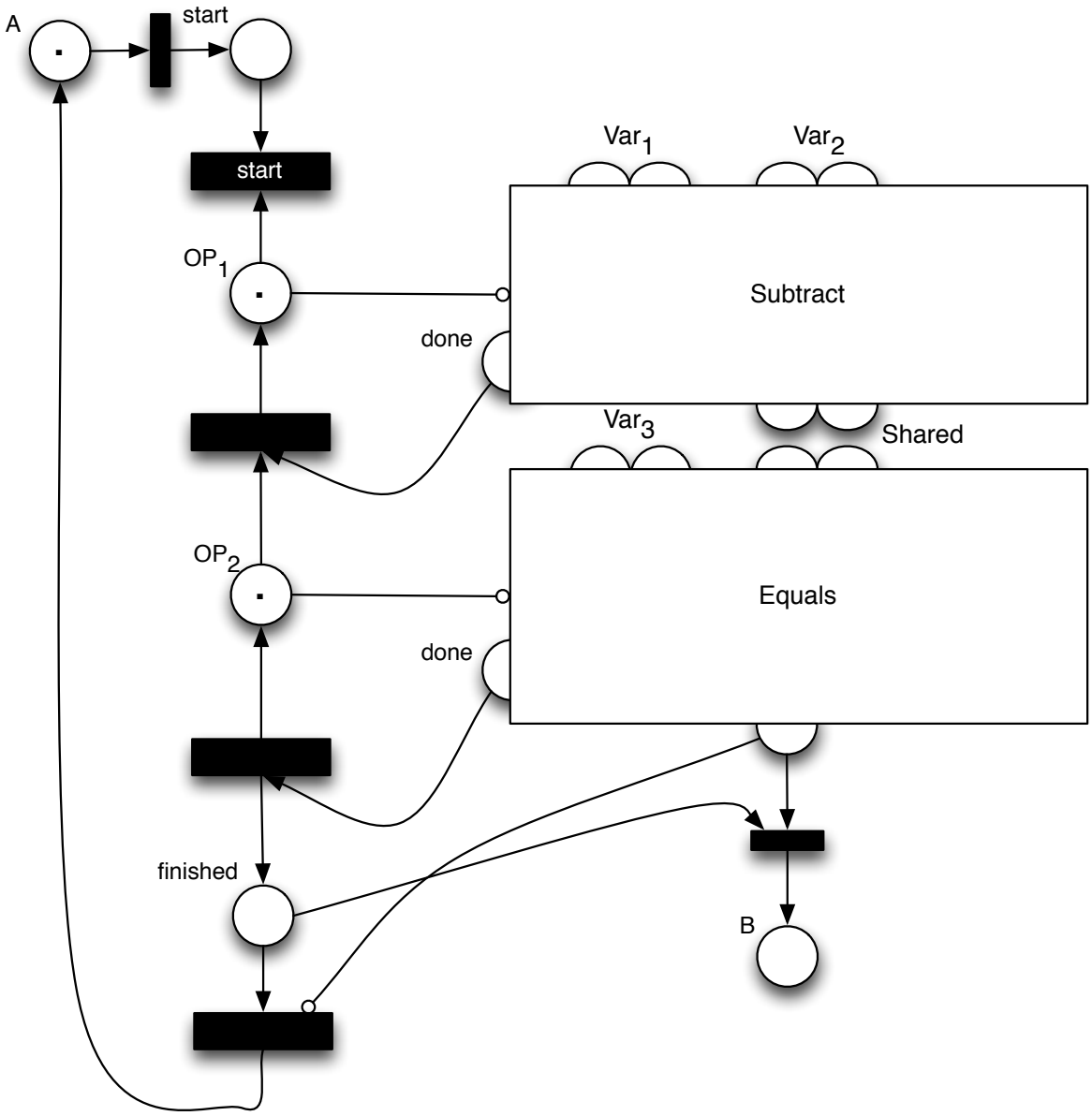

Figure 38: Inhibitor Arc Marked Petri Net converted from UML Statechart with Guard

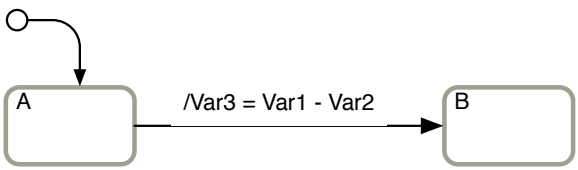

Figure 39: UML Statechart with Action

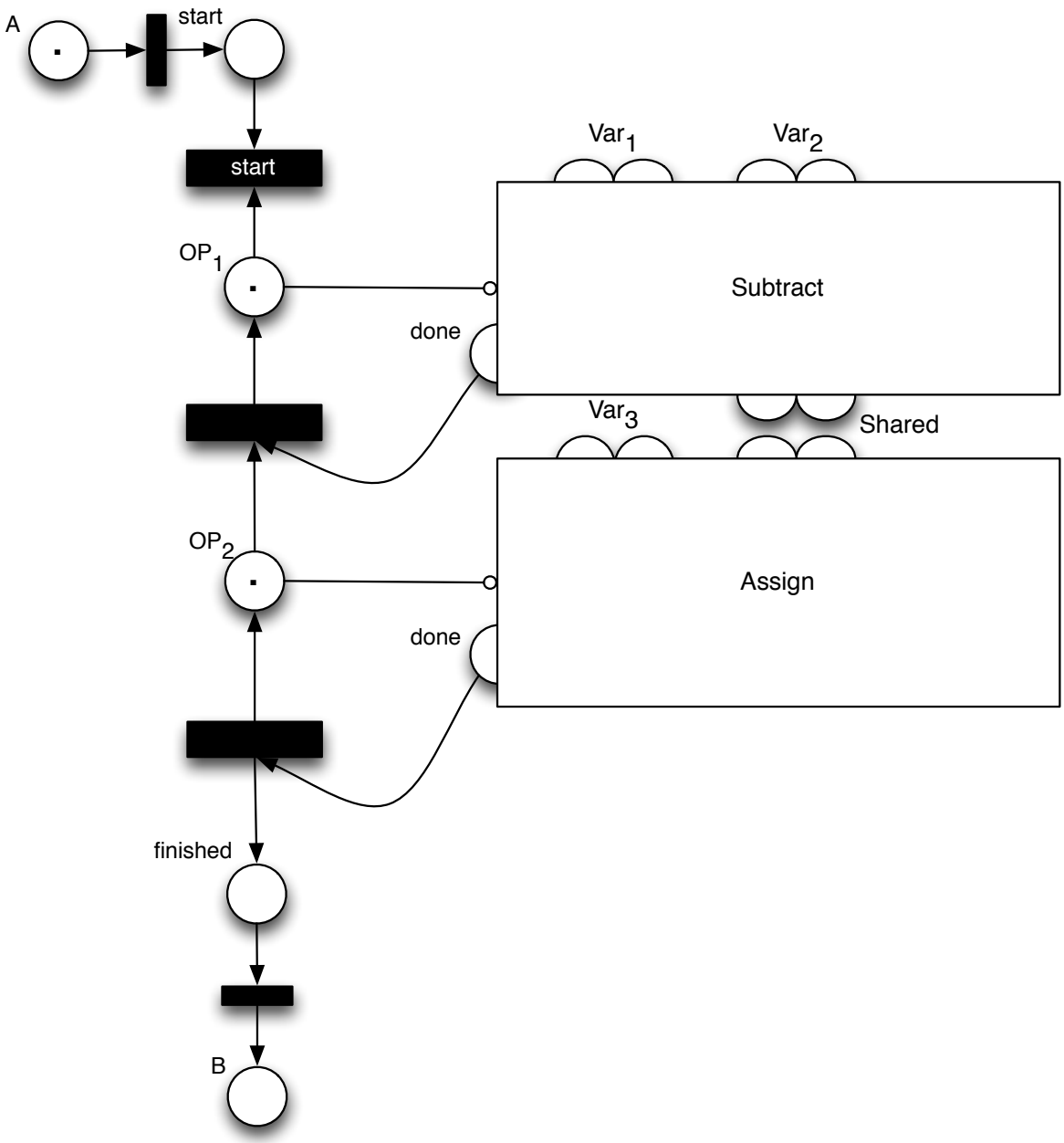

Figure 40: Inhibitor Arc Marked Petri Net converted from UML Statechart with Action

8 Transitions

The transitions listed in Figure 41 are defined in the following sections.

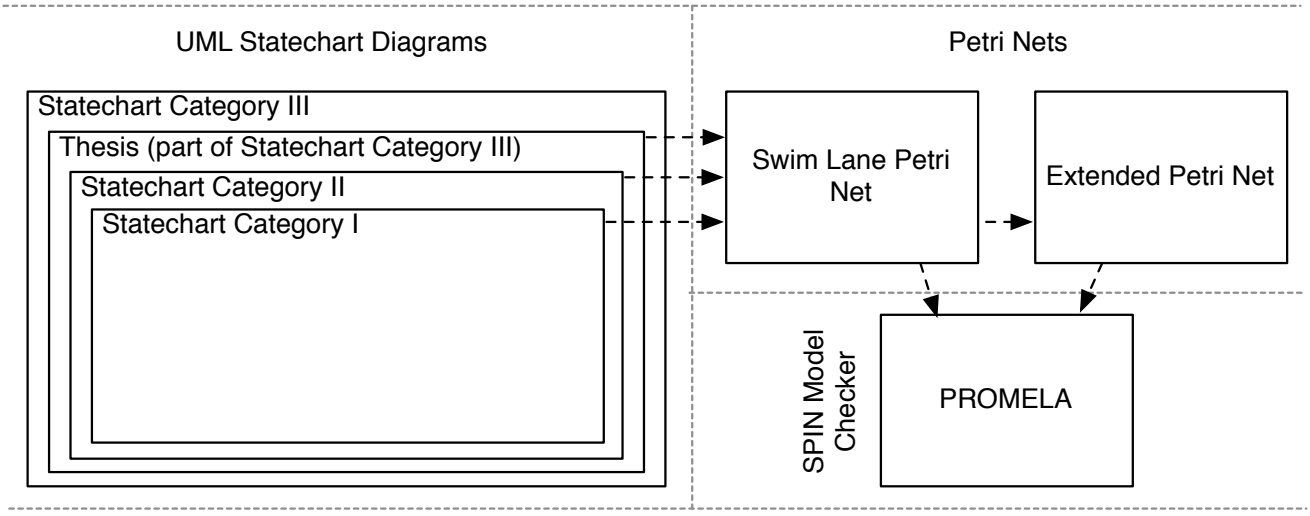

Figure 41: New Transitions

8.1 Event Driven Petri Net to Petri Net for all Extensions

Event Driven Petri Nets have two distinct differences from standard Petri nets, their input and output events. However, for the purposes of this thesis where we are interested in verification we can make two generalizations about Event Driven Petri Nets. First, inputs are equivalent to a place with an unlimited number of transitions for verification. That is, at any time there could be any number of input events. Second, outputs are equivalent to a transition firing, as no actual output needs to occur. Verification can occur on the transition firing. These generalizations are only valid for verification, and are *not* applicable for simulation.

8.2 Petri Net to PROMELA

The Petri net to PROMELA transition is accomplished by applying Algorithm 1 to a Inhibitor Arc Marked Petri net to obtain PROMELA source code.

8.3 Swim Lane Petri Net to PROMELA

The swim lane Petri net to PROMELA transition is accomplished by applying Algorithm 2 to a Swim Lane Inhibitor Arc Marked Petri net to obtain PROMELA source code.

8.4 Swim Lane Petri Net to Petri Net

The swim lane Petri net to Petri net transition is accomplished by applying Algorithm 7 to a Swim Lane Inhibitor Arc Marked Petri net to obtain an Inhibitor Arc Marked Petri net.

8.5 Statechart Category I to Petri Net

The Statechart Category I to Petri Net transition is accomplished by applying Algorithm 3 to a UML Statechart Category I to obtain an Inhibitor Arc Marked Event Driven Petri net. The Inhibitor Arc Marked Event Driven Petri net is then treated as an Inhibitor Arc Marked Petri net, where the output events are removed and the input events are replaced with Petri net places with an unlimited number of tokens.

8.6 Statechart Category II to Swim Lane Petri Net

The Statechart Category II to Swim Lane Petri Net transition is accomplished by applying all the algorithms that remove features from Category II UML Statecharts (Algorithms 4, 5, and 6), then applying Algorithms 8 and 9 to the resulting UML Statechart Category II with only Orthogonal Regions to obtain a Swim Lane Inhibitor Arc Marked Petri net.

8.7 Part of Statechart Category III to Swim Lane Petri Net

The Statechart Category III Portion to Swim Lane Petri Net transition is accomplished by applying the Statechart Category II to Swim Lane Petri Net transition, then applying Algorithm 10 to obtain a Swim Lane Inhibitor Arc Marked Petri net.

Part IV

Conclusions and Future Work

The goal of this thesis was to do two things:

- Update existing translations with a focus on traceability, and
- Create new translations that allow for a greater subset of UML Statechart Diagrams to be converted into formally analyzable forms.

There is no question that new translations have been created. The novel method of translating guards and actions with variable and expressions is a new contribution to an active research field that is intent on converting full UML Statechart Diagrams to Petri nets. Similarly, based on the listing of translations a full set have been created or re-created from UML Statecharts to various Petri nets and to PROMELA. Even traceability has been addressed by retaining the connection between UML Statechart Diagram states and Petri net places.

For example, the Statechart that was presented in the introduction was claimed to be more difficult to analyze, and that the translations presented in this thesis would simplify its analysis. The question is, is it easier to identify the connecting states after it has been converted to a Petri net. So, in Figure 42, we have the same difficult to analyze Statechart.

After applying the transformations presented in the thesis, the Statechart in Figure 42 is represented as a Petri net in Figure 43. Referring back to the original question posed in the introduction, how obvious is it that there are two transitions from H to B, the answer is significantly more clear with the Petri net representation, which is then significantly easier to analyze and validate as a PROMELA representation. This simplification of analyzable and verifiable elements becomes that much more important as models of behavior grow in size and complexity to meet the needs and demands of the real world.

There are several concepts that are natural places for future work. Certainly there is significantly more that could be said about the properties of Swim Lane Petri nets and their ability to

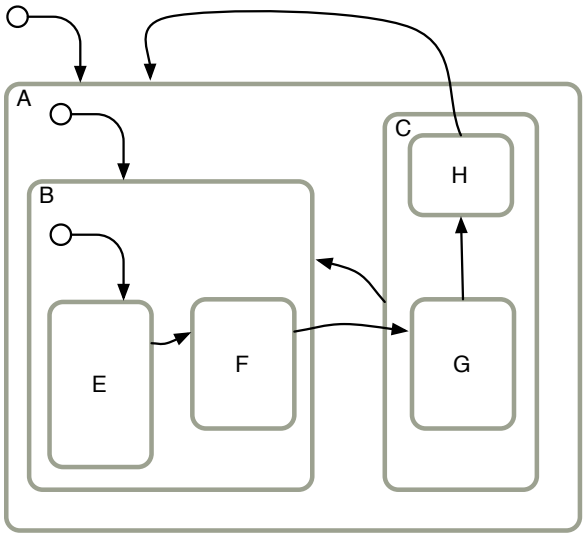

Figure 42: Difficult to Analyze Statechart

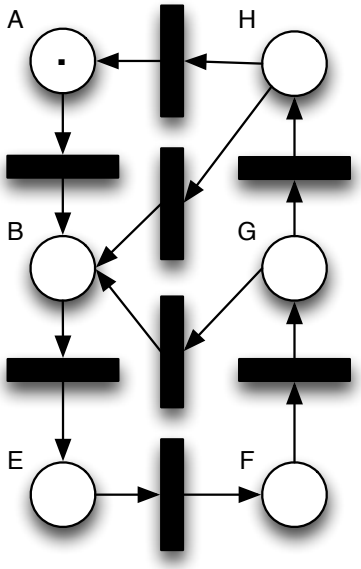

Figure 43: Easier to Analyze Petri Net translated from Figure 42

model systems of systems especially with shared resources of either transitions or places. The novel framework for converting variable based expressions used as guards and actions could be extended with new data types and more operators. Finally, there are still Category III UML Statechart Diagram features that have not been translated and would need translations before an entire UML Statechart Diagram could be translated regardless of the features used.

References

[1] Mordechai Ben-Ari, *Principles of the spin model checker*, Springer, 2008.

[2] Christine Choppy, Kais Klai, and Hacene Zidani, *Formal verification of uml state diagrams: a petri net based approach*, ACM SIGSOFT Software Engineering Notes **36** (2011), no. 1, 1–8.

[3] Gerald C Gannod and Sunil Gupta, *An automated tool for analyzing petri nets using spin*, Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on, IEEE, 2001, pp. 404–407.

[4] David Harel, *Statecharts: A visual formalism for complex systems*, Science of computer programming **8** (1987), no. 3, 231–274.

[5] David Harel, Amir Pnueli, Jeanette Schmidt, and Rivi Sherman, *On the formal semantics of statecharts*, Proc. 2nd IEEE Symp. on Logic in Computer Science (1987), pp. 54–64.

[6] G Holzmann, *The spin model checker: Primer and reference manual. addison-wesley*, Reading (2003).

[7] Zhaoxia Hu and Sol M Shatz, *Mapping uml diagrams to a petri net notation for system simulation*, 16th Int. Conf. on Software Engineering & Knowledge Engineering (SEKE 2004), 2004, pp. 213–9.

[8] ———, *Explicit modeling of semantics associated with composite states in uml statecharts*, Automated Software Engineering **13** (2006), no. 4, 423–467.

[9] Paul Jorgensen, *Modeling software behavior: A craftsman's approach*, CRC Press, 2009.

[10] Diego Latella, Istvan Majzik, and Mieke Massink, *Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker*, Formal Aspects of Computing **11** (1999), no. 6, 637–664.

[11] Diego Latella, Istvan Majzik, Mieke Massink, et al., *Towards a formal operational semantics of uml statechart diagrams*, Proceedings of the IFIP TC6/WG6, vol. 1, 1999, p. 465.

[12] Johan Lilius and Ivn Porres Paltor, *The semantics of uml state machines*, (1999).

[13] Erich Mikk, Yassine Lakhnech, Carsta Petersohn, and Michael Siegel, *On formal semantics of statecharts as supported by statemate*, Workshop, Ilkley, vol. 14, Citeseer, 1997, p. 15.

[14] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J Holzmann, *Implementing statecharts in promela/spin*, Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on, IEEE, 1998, pp. 90–101.

[15] C Pajault, J-F Pradat-Peyre, and P Rousseau, *Adapting petri nets reductions to promela specifications*, Formal Techniques for Networked and Distributed Systems–FORTE 2008, Springer, 2008, pp. 84–98.

[16] James Lyle Peterson, *Petri net theory and the modeling of systems.*, PRENTICE-HALL, INC., ENGLEWOOD CLIFFS, NJ 07632, 1981, 290, 1981.

[17] Miro Samek, *Practical uml statecharts in c/c++: event-driven programming for embedded systems*, Newnes, 2009.

[18] Timm Schäfer, Alexander Knapp, and Stephan Merz, *Model checking uml state machines and collaborations*, Electronic Notes in Theoretical Computer Science **55** (2001), no. 3, 357–369.

[19] Bran Selic, *An efficient object-oriented variation of the statecharts formalism for distributed real-time systems*, Proceedings of the 11th IFIP WG10, vol. 2, 1993, pp. 335–344.

[20] Dániel Varró, *A formal semantics of uml statecharts by model transition systems*, Graph Transformation, Springer, 2002, pp. 378–392.