PeNTa: Formal Modeling for Multi-touch Systems Using Petri Net

Francisco R. Ortega, Su Liu, Frank Hernandez, Armando Barreto,
Naphtali Rishe, and Malek Adjouadi

Florida International University, Miami FL 33199, USA
fort007@fiu.edu
http://www.franciscoraulortega/

Abstract. Multi-touch technology has become pervasive in our daily lives, with iPhones, iPads, touch displays, and other devices. It is important to find a user input model that can work for multi-touch gesture recognition and can serve as a building block for modeling other modern input devices (e.g., Leap Motion, gyroscope). We present a novel approach to model multi-touch input using Petri Nets. We formally define our method, explain how it works, and the possibility to extend it for other devices.

Keywords: Multi-touch, Petri Nets, Modern Input Devices.

1 Introduction

The importance in Human-Computer Interaction (HCI) of descriptive models (e.g., "Three-state model for graphical input"[1]) and predictive models (e.g., Fitts' law) can be seen in the seminal work by[2,3]. A descriptive model is a "loose verbal analogy and methaphore"[4], which describes a phenomenon[5]. A predictive model is expressed by "closed-form mathematical equations"[4], which predict a phenomenon[5]. We present a sound mathematical model using PetriNets for Multi-Touch systems, with the possibility of expanding it to other modern input devices (e.g., Leap Motion). Our motivation is to use a model for simulation of Multi-Touch systems and as a real-time execution system for different 3D applications, such as 3D Navigation[6]. We present **Petri Net Touch** (**PeNTa**), a modeling tool for multi-touch systems.

Input systems formalism is not recent. The pioneer work by Newman (1968) used a state diagram to represent a graphical system[7]. The seminal work by Bill Buxton in a "A Three-State Model of Graphical Input"[1] used finite-state machines (FSM). Around the same time as Buxton's work, a well-rounded model for input interactions was published by Myers[8].

Multi-touch gesture detection, or detection of touch events, have been explored. In 2013, Proton and Proton++ showed the use of Regular Expressions (RegEx) to accomplish gesture detection[9]. Lao et al[10] used state-diagrams for the detection multi-touch events. Context-free grammar was used by Kammer et al. to describe multi-touch gestures without taking implementation into consideration[11]. Gesture Coder[12] creates FSMs by demonstration to later use them to detect gestures. Gesture Works and

M. Kurosu (Ed.): Human-Computer Interaction, Part I, HCII 2014, LNCS 8510, pp. 361–372, 2014.

Open Exhibits by Ideum have a high-level language description, using XML, called GestureML (GML)[1]. A rule-based language (e.g., CLIPS) was used to define the Midas framework[13].

Petri Nets have also been used to detect gestures. Nam et al. showed how to use Coloured Petri Nets (CPN) to achieve hand (data glove) gesture modeling and recognition [14], using Hidden Markov Models (HMM) to recognize gesture features that are then fed to a Petri Net. Petri Nets have been shown to be applicable in event-driven system[15], which is another reason we are interested in them to model input devices. While our approach is based on high-level Petri Nets, Spano et al.[16,17] showed how to use Non-Autonomous Petri Nets[18], low-level Petri Nets, for multi-touch interaction. Also, Hamon et al.[15] expanded on Spano's work, providing more detail to the implementation of Petri Nets for multi-touch interactions.

In our initial approach[19] in 2013, we proposed the use of high-level Petri Nets for multi-touch interaction. In this paper, we defined a formal High-Level Petri Net for the use of Multi-Touch called Petri Net Touch (PeNTa), which is the basis for the Input Modeling Recognition Language (IRML) for input devices.

1.1 Motivation and Differences

Our target audiences for PenTA and IRML are three: (1)Software developers that would like to graphically model multi-touch interactions. (2) Framework developers[2] developers that wish to incorporate modern input to their capabilities. (3) Domain-Specific Language (DSL) developers that create solutions for domain-experts or for other developers.

There are several reasons why we preferred to use high-level Petri Nets versus other approaches. First we must consider the difference between low-level Petri Nets vs high-level Petri Nets. The major difference is that high-level Petri Nets have "complex data structures as data tokens, and [use] algebraic expressions to annotate net elements"[20]. This is similar to the difference between Assembly language versus a high-level language (e.g., Python). For the complete standard defining high-level Petri Nets see [20]. A high-level Petri Net it is still mathematically defined as the low-level petri-net but it can provide "unambiguous specifications and descriptions of applications"[20].

There are further reasons why we decided to use high-level Petri Nets for our approach. These reasons might be better understood when we place the PeNTa in the context of other existing approaches for input modeling: Proton and Proton++[9], Gesture Coder[12], and GestIT[3]. While we find that our approach offers more expressiveness and distributed capabilities, simultaneously providing a solid mathematical framework, the work offered by Proton++, Gesture Coder, and GestIT offers great insight in modeling multi-touch interaction, with different benefits that must be evaluated by the developer. We also like to note that our work is inspired by the contribution of the proponents of Proton and Proton++.

Why Petri Nets to define a multi-touch interaction? Petri Nets provide graphical and mathematical representations, that allows for verification and analysis. Thus providing

[1] `www.GestureML.org`

[2] Library and/or language developers also fit in this category.

[3] Which also refers to Hamon et al.[15].

a formal specification that can be executed. Petri Nets also allow distributed systems to be represented. Finally, a finite state machine can be represented in a Petri-Net but a Petri-Net may not be represented as a FSM. In other words, Petri Nets have more expressive representational power.

Proton and Proton++ offer a novel approach to multi-touch interaction using RegEx. Such approach offers an advantage to those who understand regular expressions. However, there may be some disadvantages in using RegEx for our goals. Expressions of some gestures can be lengthy, such as the scale gesture[9]: $D_1^s M_1^s * D_2^a (M_1^s | M_2^a) * (U_1^s M_2^a * U_2^a | U_2^a M_1^s * U_1^s)$[4]. Spano et al.[17] present additional differences between Petri Nets and the use of regular expressions in Proton++. Another potential disadvantage of using regular expressions, is that some custom gestures may become harder to represent.

Gesture Coder offers a great approach to create gestures by demonstration. While we don't follow this approach, PeNTa could also created the Petri Nets by machine learning training. The representation of the training for Gesture Coder results in a FSM. FSMs can become large and do not provide the expressive power and distributed representation of Petri Nets.

We have also looked at GestIT. This is the closest approach to ours. GestIT uses low-level petri nets. While this work represents a valuable contribution, it may lack the expressiveness of using data structures in tokens which are part of high-level Petri Net. We find, nonetheless, that GestIT can provide some great ideas to further improve our approach in PeNTa and IRML.

The approach of GestIT is similar to Proton++ given that the set of points (trace) is broken down into points and the gesture becomes a pattern of those points. While the expressiveness of our HLPN allows to use this approach, we have preferred to define the each token as an individual trace.

Our work includes the novel approach to use high-level Petri Nets for multi-touch recognition including the definition of our HLPN. The fact that we are using Petri Net allows for formal specifications to be applied to multi-touch, and perhaps other modern input devices (e.g., Leap Motion), and enables a distributed framework while keeping the approach simple (in comparison to low-level Petri Nets). This means that by encapsulating complex data structures in tokens and allowing for algebraic notation and function calling in the transitions of a Petri Net, the modeling is not restricted to one individual approach. Furthermore, the data structure kept in each token, can maintain history information that may be relevant to some processes. This will be explained in detail in the following two sections.

2 HLPN: High-Level Petri Nets and IRML

For our model, we define our High-Level Petri Net[21,22], as $HLPN = (N, \Sigma, \lambda)$. This contains the Net **N**, the specifications Σ and the inscription λ.

The Net **N** is formed with places **P**, transitions **T**, and connecting arc expressions (as functions **F**). In other words, a Petri-Net is given by a three-tuple $N = (P, T, F)$, where $F \subset (P \times T) \cup (T \times P)$. Petri Nets' arcs can only go from **P** to **T** or **T** to **P**. This

[4] D=Down, M=Move, U=Up; s=shape, b=background, a=any (a—b).

can be formally expressed stating that the sets of places and transitions are disjoint, $P \cap T = \emptyset$. Another important characteristics of Petri Nets is that they use multi-sets[5] (elements that can be repeated) for the input functions, $(I : T \rightarrow P^{\infty})$ and output functions, $(O : P \rightarrow T^{\infty})$[23].

The specification Σ represents the underlying representation of tokens. This is defined as $\Sigma = (S, \Omega, \Psi)$. The set S contains all the possible token[6] data types allowed in the system. For our particular case, our data type is always the same, which is a multi-touch token **K**, as shown in Table 1. The set Ω contains the token operands (e.g., plus, minus). The set Ψ defines the meaning and operations in Ω. In other words, the set Ψ defines how a given operation (e.g., plus) is implemented. For our case, we use regular math and boolean algebra rules, without the need to redefine. This is the default for **PeNTa** tokens. It is important that all the tokens in our HLPN are the signature of (S, Ω).

The inscription λ defines the arc operation. This is defined as $\lambda = (\phi, L, \rho, M_0)$. The data definition represented by ϕ is the association of each place $p \in P$ with tokens. This means that **places** should accept only variables of a matching data type. In our case, we have token **K**, which represents the multi-touch structure. The inscription also has labeling **L** for the arc expressions, such as $L(x,y) \iff (x,y) \in F$. For example, a transition that goes from place B to transition *4* will be represented as $L(B,\ 4)$. The operation of a given arc (function) is given by $\rho = (Pre, Post)$. This are well-defined constraint mappings associated with each arc expression, such as $f \in F$. The **Pre** condition allows our HLPN to enable the function, if the boolean constraint evaluates to true. Then, the **Post** condition will execute the code if the function is enabled (ready to fire). Finally, the initial marking is given by M_0, which states the initial state of our HLPN.

2.1 Dynamic Semantics

In order to finalize the formal definition of our HLPN, we include some basic notes about the dynamic aspects of our Petri Net. First, markings of a HLPN are mappings $M : P \rightarrow Tokens$. In other words, places map to tokens. Second, given a marking M, a transition $t \in T$ is enabled at marking M iff $Pre(t) \leq M$. Third, given a marking M, α_t is an assignment for variables of t that satisfy its transition condition, and A_t denotes the set of all assignments. Define the set of all transition modes to be $TM = \{(t, \alpha_t) \,|\, t \in T, \alpha_t \in A_t\} \iff Pre(TM) \leq M$. An example of this definition is a transition spanning multiple places, as shown in Figures 1 and 2 (concurrent enabling). Fourth, given a marking M, if $t \in T$ is enabled in mode α_t, firing t by a step may occur in a new marking $M' = M - Pre(t_{\alpha_t}) + Post(t_{\alpha_t})$; a step is denoted by $M[t > M'$. In other words, this is the transition rule. Finally, an execution sequence $M_0[t_0 > M_1[t_1 > ...$ is either finite when the last marking is terminal (no more transitions are enabled) or infinite. The behavior of a HLPN model is the set of all execution sequences, starting from M_0.

[5] Also known as Bag Theory.

[6] Some Petri Nets' publications may refer to tokens as "sorts".

2.2 Multi-touch

A multi-touch display (capacitive or vision-based) can detect multiple finger strokes at the same time. This can be seen as a finger trace. A **trace** is generated when a finger touches down onto the surface, moves (or stays static), and it is eventually lifted from it. Therefore, a trace is a set of touches of a continuous stroke. While it is possible to create an anomalous trace with the palm, for example, we are only taking into consideration normal multi-finger interaction. However, our data structure (explained in detail later) could be modified to fit different needs, including multiple users and other sensors that may enhance the touch interaction. Given a set of traces, one can define a **gesture**. For example, a simple gesture may be two fingers moving on the same path, creating a **swipe**. If they are moving in opposite ways (at least one of the fingers), this can be called a **zoom out** gesture. If the fingers are moving towards each other, then this is a **zoom in** gesture. A final assumption we make for the multi-touch system is the following: if a touch interaction is not moving, it will not create additional samples but increment the holding time of the finger. Note that this is not inherently true in all multi-touch systems. For example, in native WINAPI (Windows 7) development, samples are generated, even if the finger is not moving, but holding. We filter those samples out by creating a small threshold that defines that the finger is not moving (even if it may be moving slightly).

2.3 Arc Expressions

Each arc is defined as a function **F**, which is divided into two subsets of inputs **I** and outputs **O**, such that $F = I \cup O$. In the inscription ρ of this HLPN, the arc expression is defined as **Pre** and **Post** conditions. Simply put, the **Pre** condition either enables or disables the function, and the **Post** condition updates and executes call-back events, in our HLPN.

Each function **F** is defined as $F = Pre \cup Post$, forming a four-tuple $F = (B,U,C,R)$, where **B** and **U** are part of the **Pre** conditions, and **C** and **R** are the **Post** conditions. **B** is the boolean condition that evaluates to true or false, **R** is the *priority function*, **C** is the call-back event, **U** is the update function.

The boolean condition **B** allows the function to be evaluated using standard boolean operators with the tokens, in C++ style (e.g., $T_1.state == STATE.UP$). If no condition is assigned, the default is ***true***. The priority function **R** instantiates a code block, with the purpose of assigning a priority value to the arc expression. The call-back event **C** allows the Petri Net to have a function callback with conditional *if* statements, local variable assignments, and external function calling. This can help to determine which function should be called. If no callback event is provided, then a default $genericCallBack(Place\ p, Token\ t)$ is called. The update function **U** is designed to bring the next touch sample using $update(T_1)$, or setting a value to the current sample of a given token using $set(T_1,\ TYPE.STATE,\ STATE.UP)$.

Places and transitions in our HLPN have special properties. This is true for **P**, which has three types: *initial*, *regular*, and *final*. This allows the system to know where tokens will be placed when they are created (*initial*) and where the tokens will be located when they will be removed (*final*).

Table 1. Multi-Touch Data Structure

Name	Description
id	Unique Multi-Touch Identification
tid	Touch Entry Number
x	X display coordinate
y	Y display coordinate
state	Touch states (DOWN, MOVE, UP)
prev	Previous sample
get(Time t)	Get previous sample at time **t**
tSize	Size of sample buffer
holdTime	How many milliseconds have spawn since last rest
msg	String variable for messages

Picking the next transition or place for the case when there is one input and one output is trivial (the next **T** or **P** is the only available). However, when there are multiple inputs or outputs, picking the next one to check becomes important in Petri Net implementation[24]. Our "picking" algorithm is a modified version of the one by Mortensen [24]. Our algorithm combines the random nature found in other CPN[25] selection and the use of *priority functions*. Our algorithm sorts the neighboring **P** or **T** by ascending value, given by the priority function **R**, and groups them by equivalent values, (e.g., $G_1 = 10, 10, 10$, $G_2 = 1, 1$). The last **P** or **T** fired may be given a higher value if found within the highest group. Within each group, the next **P** or **T** is selected randomly.

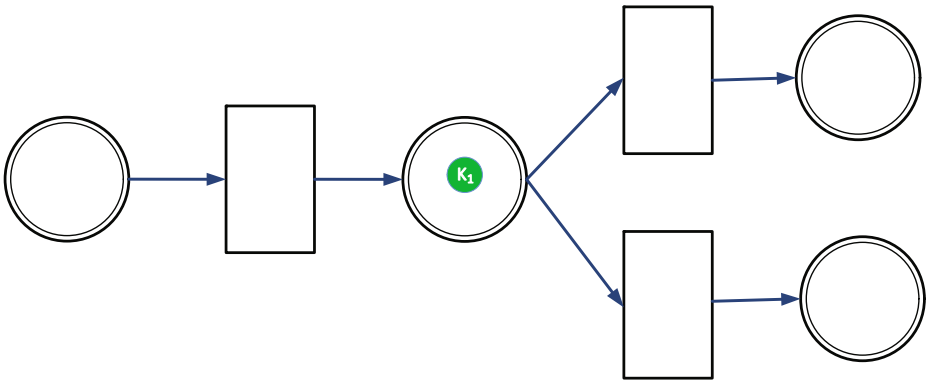

Fig. 1. Parallel PN: State 1

2.4 Tokens and the Structure

A powerful feature of Petri Nets is their discrete markings, called **Tokens**. This feature allows the marking of different states and the execution of the Petri Net. When tokens go through an input function **I** or output functions **O**, they are consumed. For our particular modeling, we use the token as the representation of a data structure, as shown in Table 1. Each token translates to a **trace**. Furthermore, this data structure contains the current sample and a buffer of previous samples(**touches**).

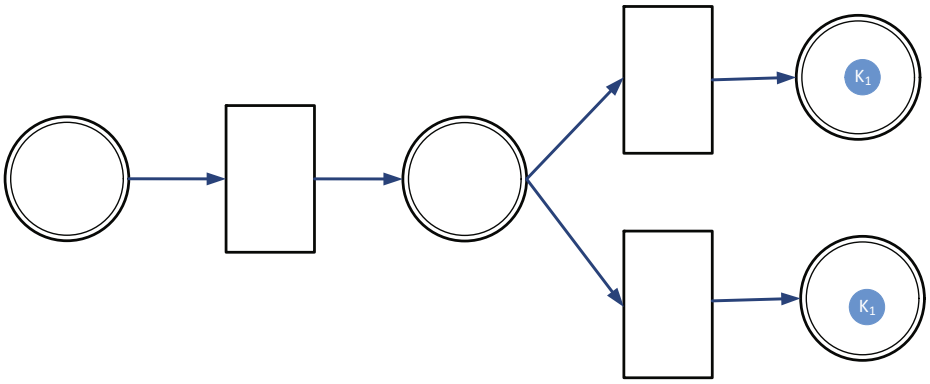

Fig. 2. Parallel PN: State 2

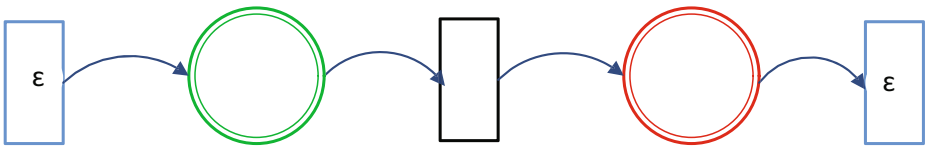

Fig. 3. Cold transitions (Entry and Exit)

When tokens are consumed into a transition, the **Post** condition creates a new token. If the desired effect is of a parallel system, as shown in Figure 1 , then a transition can create n number of tokens based on the original token. In Figure 2, token K_1 is cloned into two identical tokens, both called K_1. To represent the new tokens, different colors were choosen in Figures 1 and 2.

The only token data type for PeNTa in our HLPN is a multi-touch structure, type **K**, as shown in Table 1. The identification given by a system is denoted as **id**. Depending on the system, this may be a unique number while the process is running (integer long) or as a consecutive integer starting from $1 \ldots m$, lasting the through the duration of the gesture performed by the user. The latter definition is also contained in the variable **tid**. Display coordinates are given by **x** and **y**. The **state** variable represents the current mode given by "DOWN", "MOVE", "UP". The **holdTime** helps to determine how many milliseconds have lapsed since the user has not moved from the current position. Our data structure assumes that a new sample only happens when the user has moved beyond a threshold. Finally, this data structure acts as a pointer to previous history touches in a buffer of size η. Therefore, the buffer (if it exists,) can be accessed with the function $get\,(Time\,\iota)$ or the previous sample with the variable **prev**.

In Petri Nets or HLPN, at least one initial Marking M_0 needs to be known, especially for analysis. In the case of simulations (which is discussed later), this is not a problem. However, for the case of real-time execution, which is our primary purpose, the initial marking is empty. Remember that the initial marking is the place of tokens in any of the given available places. This is solved by the concept of hot and cold transitions[26], represented by ε. A hot transition does not require user input. A cold transition requires

user (or external system) input. Therefore, in the implementation of our HLPN, we define an entry cold transition and an exit cold transition, as shown in Figure 3.

Table 2. Transitions

Arc	From	To	Condition	Token Count
1	A	Down	K.state == DOWN	1
2	Down	B	update(T)	1
3	B	Move	K.state == MOVE	1
4	B	UP	K.state == UP	1
5	Move	C	update(T)	1
6	C	UP	K.state == UP	1
7	C	Move	K.state == MOVE	1
8	C	Move	update(T)	1
9	C	Zoom	K.state == MOVE && IsZoom(K_1, K_2)	2
10	Zoom	C	Update(K_1, K_2)	2
14	Swipe	C	K.state == MOVE && IsSwipe(K_1, K_2)	2
15	Swipe	D	Update(K_1, K_2)	2
17	UP	E	K.state == UP	1
18	UP	E	K.state == UP	1
19	E	Terminate	true	1

3 PeNTa: Modeling with Petri Nets

3.1 User Interface

It is important that our model is mathematically sound. However, it is also very important that it is easy to work with **PeNTa** and our HLPN. While there are great tools for Petri Nets (e.g., CPN Tools) and great frameworks for Domain-Specific Languages (DSL) (e.g., Microsoft Visual Studio DSL), we found that **PeNTa** required its own Graphical User Interface (GUI) to allow for design of input interaction, the ability for HLPN simulation and code-generation for execution. This is the reason that we decided to start developing with the Qt Framework, as well as using well-established Petri Net algorithms, such as [24]. The user interface will serve as a GUI for developers to test their input. While at this point, the model has being designed for multi-touch input, it is foreseeable that this can be expanded to other modern input types.

3.2 Language Choice

CPN uses ML general-purpose language, and it is a very popular HLPN. However, we decided to use a subset of the C++ language to define part of our PeNTa. In a recent information survey in the 3DUI mailing list, C++, C# and Java were the languages of choice in the field of 3D User Interfaces. In a formal survey, Takala et al.[27] said

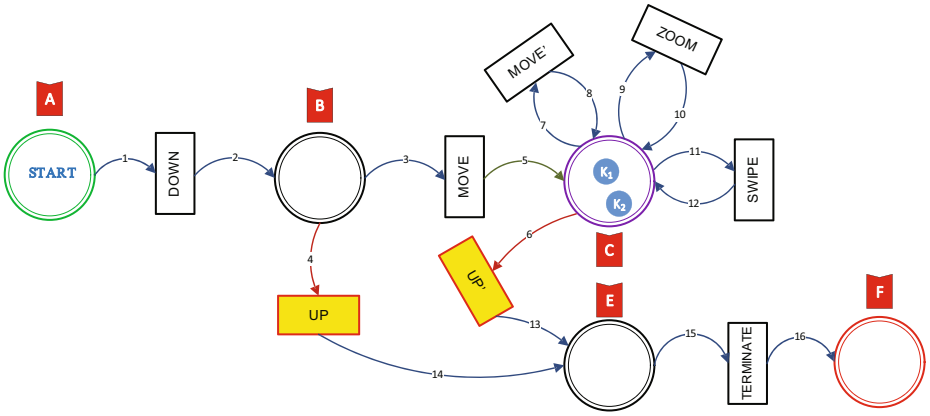

Fig. 4. Multiple Gestures in PeNTa

that the most common programming language was C++. Nevertheless, PeNTa could be implemented in any programming language. The user interface is under development at the time of this writing, and once ready, it will be uploaded into our modeling Web site[7].

3.3 A Tour of PeNTa

A tour of our model is needed to better explain how **PeNTa** works. We start with an example that deals with two gestures using a two-finger interaction: swipe and zoom. A swipe implies that the user moves two fingers in any direction. Therefore, the callback required is always the same, which is a function that reports the direction of the swipe. In the case of the zoom, zoom-in and zoom-out could be modeled separately or together. We chose the latter for our example, shown in Figure 4. This allows us to show how our model can handle gestures that require different call backs. For example, if the user is zooming-in, then it will call a zoom-in only function, otherwise, it will call a zoom-out function.

The example shown in Figure 4 is created for two-finger gestures. The figure has places, arcs, transitions and and two tokens (K_1, K_2), representing two active traces in place **C**. For this particular example, we have added letter labels to places, numbers to the arcs and names to the transitions; however, those are not part of a Petri Net. In addition, Table 2 shows each arc expression with their boolean condition and the tokens that are required to fired (e.g, two tokens). The system starts with a empty initial marking (no tokens), while it waits for the user input. Once the user touches down onto the surface, tokens are created (using the cold transitions) and placed in the START place. Given that the tokens will start with a "DOWN" state, they will move from place **A** into place **C**, using transitions 1 and 2. The first arc just consumes the token, and arc *2* updates the token with the next sample into place **B**. This is done one token a time, by design. The reason for this is that if it required two tokens, the system may starve at this

[7] `openrml.com`

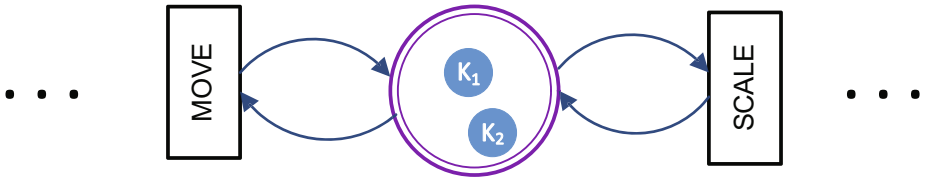

Fig. 5. Partial Petri Net for Scale

point. Once in place **B**, since the token was updated to the next sample, the Petri Net has to decide where to send the token. It has two options, either arc 3 or arc 4. Assuming that the state is "MOVE", now each token gets moved into place **C** with an updated touch sample. Now, we are at the point shown in Figure 4. This HLPN now has to decide where to go next. This is where the picking algorithm explained earlier comes into play. For this example, **MOVE´**, **ZOOM**, **SWIPE**, and **UP** have priority function values 1, 10, 10, and 2, respectively. This means the group with **ZOOM** and **SWIPE**, will be the first to be checked for constraints, since they have the highest values. The HLPN will randomly pick either one and check if it can be enabled (fired) based on the boolean condition. Assume, for example, that it picks **SWIPE** and the boolean condition is *true*. It will be *true* if two tokens are in place **C**, both tokens are in state "MOVE", and the function **isSwipe**, which is an external C++ method, are true. If this is true, then it will call back a swipe function, passing a copy of the token data, and then update to the next sample via arc 12. This finally brings back both tokens into place **C**. At some point, the tokens will have state "UP", and they will move to place **E** and place **F**. The reason for having two places for termination is to allow the designer to choose when to make the final termination. The final state will send two no-longer usable tokens into an **exit** cold transition (not shown in the figure), to be destroyed by the execution system.

While we presented in Figure 4 a single Petri Net that models various gestures, it is possible to create individual Petri Nets as shown in Figure 5, and combine them in a set of Petri Nets. For example, individual Petri Nets PN_i can form a model P = $(PN_1, PN_2, PN_3, ...PN_n)$, where once constructed, the model can be executed. Each PN_i can run in parallel and disabled itself when the condition is not met.

3.4 Simulation and Execution

Petri Nets could be used for analysis (e.g., Linear Temporal Logic), simulations and execution. We have concentrated **PeNTa** for simulation and execution, as we find it the most critical aspect needed for input devices. Nevertheless, analysis could be implemented into PeNTa.

There are different ways to simulate **PeNTa**. Non-user-generated data could be used for the simulation, providing an initial marking with a history of the possible samples. Another option it is to record the user interaction creating tokens and a buffer to feed those tokens. There are multiple ways to go about this, but we used MySQL to store the data. This is a very useful way to test new gestures, create custom gestures, and troubleshoot problems with an existing system.

Execution is the primary purpose of PeNTa. Given a well-defined model, this can run in real-time, using the definitions of our HLPN. As stated before, our HLPN would need additional **cold** transitions (entry and exit), since this is user-generated input.

4 Conclusion

We presented a novel modeling approach to multi-touch interaction, named **PeNTa**. This is given by our High-Level Petri Net, defined with input systems in mind. We also showed an example of a model. In the near future, we will complete the GUI tool to design, execute and simulate **PeNTa**. We will also look into the use of other input devices, as well as combining our HLPN model with multiple devices in one design.

Acknowledgements. This work was sponsored by NSF grants NSF-III-Large-MOD 800001483,HRD-0833093, CNS-0959985, CNS-0821345, and CNS-1126619. Francisco Ortega is a GAANN fellow (US Department of Education) and McKnight Dissertation Fellow (Florida Education Fund).

References

1. Buxton, W.: A three-state model of graphical input. In: Human-computer Interaction-INTERACT 1990, pp. 449–456 (1990)
2. English, W.K., Engelbart, D.C., Berman, M.L.: Display-Selection Techniques for Text Manipulation. IEEE Transactions on Human Factors in Electronics (1), 5–15 (1967)
3. Gray, W.D., John, B.E., Atwood, M.E.: Project Ernestine: Validating a GOMS analysis for predicting and explaining real-world task performance. Human–Computer Interaction 8(3), 237–309 (1993)
4. Pew, R.W., Baron, S.: Perspectives on human performance modelling. Automatica 19(6), 663–676 (1983)
5. Mackenzie, I.S.: Human-Computer Interaction. An Empirical Research Perspective. Newnes (December 2012)
6. Bowman, D.A., Kruijff, E., LaViola Jr., J.J., Poupyrev, I.: 3D user interfaces: theory and practice. Addison-Wesley Professional (2004)
7. Newman, W.M.: A system for interactive graphical programming, pp. 47–54 (1968)
8. Myers, B.A.: A new model for handling input. ACM Transactions on Information Systems (TOIS) 8(3), 289–320 (1990)
9. Kin, K., Hartmann, B., DeRose, T., Agrawala, M.: Proton++: a customizable declarative multitouch framework. In: UIST 2012: Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology, ACM Request Permissions (October 2012)
10. Lao, S., Heng, X., Zhang, G., Ling, Y., Wang, P.: A gestural interaction design model for multi-touch displays, pp. 440–446 (2009)
11. Kammer, D., Wojdziak, J., Keck, M., Groh, R., Taranko, S.: Towards a formalization of multi-touch gestures. In: ITS 2010: International Conference on Interactive Tabletops and Surfaces, ACM Request Permissions (November 2010)
12. Lü, H., Li, Y.: Gesture coder: a tool for programming multi-touch gestures by demonstration, pp. 2875–2884 (2012)
13. Scholliers, C., Hoste, L., Signer, B., De Meuter, W.: Midas: a declarative multi-touch interaction framework, pp. 49–56 (2011)

14. Nam, Y., Wohn, N., Lee-Kwang, H.: Modeling and recognition of hand gesture using colored Petri nets. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans 29(5), 514–521 (1999)
15. Hamon, A., Palanque, P., Silva, J.L., Deleris, Y., Barboni, E.: Formal description of multitouch interactions. In: EICS 2013: Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, ACM Request Permissions, New York, USA, pp. 207–216 (June 2013)
16. Spano, L.D., Cisternino, A., Paternò, F.: A compositional model for gesture definition. In: Winckler, M., Forbrig, P., Bernhaupt, R. (eds.) HCSE 2012. LNCS, vol. 7623, pp. 34–52. Springer, Heidelberg (2012)
17. Spano, L.D., Cisternino, A., Paternò, F., Fenu, G.: GestIT: a declarative and compositional framework for multiplatform gesture definition. In: EICS 2013: Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, ACM Request Permissions, New York, USA, pp. 187–196 (June 2013)
18. David, R., Alla, H.: Discrete, Continuous, and Hybrid Petri Nets. Springer (November 2010)
19. Ortega, F.R., Hernandez, F., Barreto, A., Rishe, N.D., Adjouadi, M., Liu, S.: Exploring modeling language for multi-touch systems using petri nets. In: ITS 2013: Proceedings of the 2013 ACM International Conference on Interactive Tabletops and Surfaces. ACM (October 2013)
20. High-level Petri Nets-Concepts: Definitions and graphical notation. Final Draft International Standard ISO/IEC 15909 (2000)
21. Genrich, H.J., Lautenbach, K.: System modelling with high-level Petri nets. Theoretical Computer Science 13(1), 109–135 (1981)
22. Liu, S., Zeng, R., He, X.: PIPE-A Modeling Tool for High Level Petri Nets (2011)
23. Peterson, J.L.: Petri net theory and the modeling of systems. Prentice Hall (1981)
24. Mortensen, K.H.: Efficient data-structures and algorithms for a coloured Petri nets simulator, pp. 57–74 (2001)
25. Jensen, K., Kristensen, L.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Springer (1996)
26. Reisig, W.: Petri Nets. An Introduction. Springer (July 2012)
27. Takala, T.M., Rauhamaa, P., Takala, T.: Survey of 3DUI applications and development challenges, pp. 89–96 (2012)