PyTri, a Visual Agent Programming Language

Jochen Simon and Daniel Moldt

University of Hamburg, Department of Informatics
http://www.informatik.uni-hamburg.de/TGI/

Abstract. PyTri is a Python based visual agent programming language which has been designed top-down to utilize the possibilities of graphical representation of control flow by amending the concepts of Petri nets. Its main inspiration, Mulan, which is based on Java nets, originated bottom-up from a powerful formalism, which allows modeling and programming multi-agent systems. The here presented PyTri vision uses multiple types of transitions and places, with a specialized Python-based inscription language, in order to offer a rich semantics that allows expressive and compact representation of executable code. GUI widgets can be directly embedded into the nets and can infuse them with tokens upon user interaction.
PyTri aims to support coarsening mechanisms, instead of the net within net paradigm, and includes a special place class that allows representation of independent control flow streams. It strives to enable modularization of complex multi-agent applications as one huge flat structure, not requiring them to be separated into discrete layers. Integral to PyTri is a future tool for visualization and manipulation which is tailored to the needs of programmers using the language.

Keywords: PyTri, Renew, graphical visual programming language, agents, Petri nets, formalism, reactive programming, Python

1 Introduction

Visual Languages offer more modalities[5] for representing and incorporating the semantics of a program, compared to traditional textual languages. They directly express the control flow of the processes inside an application, unlike line-based text files with procedural jumps. In recent functional languages a plethora of different ways to compose functions emerged, resulting in arrows which are more general than monads. However, understanding the underlying mechanisms and being able to utilize them, requires at least basic proficiency in category theory.

Petri nets offer a very intuitive way of representing arbitrary computation processes by decomposing them into atomic actions and intermediate chunks of data, incorporated by transitions and places. By embedding external triggers into a net, reactive programming can be done without the hassle of splitting up the code into steps.

The human mind in general is very good at memorizing structures visually, grouping related objects by spatial proximity. There are several types of UML

diagrams which are a popular method to visualize the interrelations of the classes and objects of an application. By modeling an application with a Petri net like net formalism in one single huge structure, the big picture can be shown as a direct map of the internal processes, possibly even in three dimensions. Of course, mechanisms of coarsening have to be used, in addition to modularizing the code into organ-like fragments, or none would be able to cope with the complexity.

In [10] several existing visual languages were looked at, some general criteria were investigated and PyTri, a visual agent programming language based on Python was devised. Amongst the investigated languages were YAWL[9], AgentSheets[8], AUML[2], LabVIEW[6] and Quartz Composer[1]. PyTri is heavily inspired by the Java net formalism of Renew, which is based on reference nets, whose theoretical foundations were laid down in [7]. Java nets allow modeling and execution of complex multi-agent based distributed applications[3], through a sophisticated tool set[4]. Java nets have been designed bottom-up, starting with the reference net formalism, and are especially suited for modeling.

In Section 2 the mission statement for PyTri, the aims of its semantics, and the most important properties of its formalism are given. Section 3 introduces the idea of specialized types of places and transitions, a mixed class of elements, PyTri's focus on coarsening and refinement, and tool support. Section 4 discusses the results of the design of PyTri and future work.

2 Goals of PyTri

PyTri aims to be an expressive and compact language, condensing common programming tasks to a clean and descriptive representation. PyTri has been designed from the start with the goal of making the creation of multi-agent based applications as comfortable as possible. Essential to PyTri is an extensive graphical editor displaying the net code, designed for developing, debugging and manipulating. The concepts of PyTri have been designed top-down with a focus on programming, not modeling. PyTri aims to employ a rather complex semantics, and is targeted at experienced programmers willing to learn its large vocabulary. It still tries to be intuitive, without dumbing down, in order to be accessible to novices, given a set of good tutorials, gradually introducing the concepts.

The Java net formalism has some assembler like aspects to it, because one has to perform a lot of bookkeeping by hand to perform basic tasks. The formalism of PyTri attempts to take graphical programming to a higher level of abstraction, retaining a semantics that is very similar to that of Petri nets, but differing in several important ways, still including elements that act very much like traditional places and transitions. Rather than having one kind of place and transition that is used for everything, the formalism is based on specialized elements that are the basic primitives of the language and can be compared to statements in textual languages, like `if`, `while`, `try`, `def` or `return`.

In order to allow compact branching of control flow, the PyTri formalism deviates from the Petri net semantics by allowing transitions to dynamically decide to suppress the donation of tokens to specific target places. This means

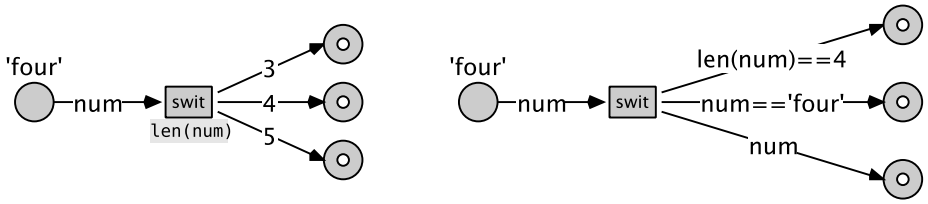

Fig. 1. Two variants of the switch transition used to decide control flow

that the actual control flow is no longer apparent from the arc connections alone, but dependent on the inscriptions of the transitions. Such conditional branching would suffice, and may be the best solution in some cases, but in other cases having a specialized transition type, with a different inscription semantics, can be useful syntactic sugar and convey the intention of the code in a better way. An example is the Switch transition shown in Figure 1, which can be used to evaluate an expression and donate a token only through the arc whose inscription matches, or can evaluate a set of arcs inscribed with boolean expressions and independently donate tokens through all those that are true.

PyTri's inscription language is based on Python, which was chosen because it is a very compact and powerful high-level language and offers a great standard library. It is attempted to allow almost arbitrary Python code in the inscriptions, while amending the inscription language with the required new concepts, pythonically. It uses pythonic duck typing, where Java nets guarantee that only certain types of objects can traverse an arc. A lot of the expressiveness of Java nets comes from unification, which allows fully dynamic connections between elements. PyTri aims for a simpler and easier to understand semantics, which introduces other mechanisms to cope with the loss of the prolog-like power.

3 Complex Elements and Tool Support

Place Types. Just like in Petri nets and Java nets, places are passive and do not initiate actions by themselves, however, they are preconditions to the transitions that accept tokens from them. Instead of having one general kind of place which is used universally to store data, PyTri offers several types. Usually, places are either used to store a single token or to store a collection of tokens, rarely both.

When storing single tokens, places are usually used either for signifying and determining the control flow, or as a single variable holding intermediate or persistent data objects, which leads to the place types Flow and Variable.

When storing a collection of tokens, one usually wants it to behave like one common abstract data type, depending on the context. By incorporating the required behavior in a place type, Queue or Stack for example, the programmer can be spared of the otherwise necessary manual bookkeeping. Also, special place types can exist simply to provide helpful eye candy to the developer, or simplify debugging of agents by easing the assignment of the states an agent can be in.

Transition Types. Many abstract mechanisms integral to programming are not atomic, but are composed of multiple steps. By having basic elements that execute multiple atomic phases when needed, while storing internal data, the language becomes more compact in its expressiveness. Those elements of this kind, which in essence are transition-bordered components, can be considered similar enough in concept to transitions to justify raising them to their level. Through this the concept of transitions is extended to a set of basic graphical statements building the core language. Since the individual phases are still atomic, complex multi-phase transitions should be decomposable to nets of places and atomic transitions. This decomposition however can also be virtual, since the behavior may be implemented through pure Python objects with a certain interface.

Some transition types require special interactions with certain types of places in order to achieve their purpose. Some transitions work like components built out of elements connected to other transitions via virtual arcs. PyTri has been designed from the start with the aim of creating a language tailored to the creation of multi-agent systems. Amongst the basic transitions are elements that allow sending messages to other agents, receiving messages, or a combination.

Gadgets. In order to directly embed GUI components into the nets, a new general type of element was devised. Gadgets are the representation for mixed-border components, which can perform proactive actions, but also expose their internal data to connected transitions, like a place. The internal virtual elements of a gadget are connected to the elements around it, based on the aliases of the connecting arcs. Gadgets have a consistent semantics that enables them to directly interact with the surrounding net by donating tokens, acting as a precondition, and receiving tokens. For GUI components, the behavior can be encapsulated in pure Python proxy objects that perform all required bookkeeping transparently. In Figure 2, a dialog is visible and tangible to a local user, as long as a token is inside the leftmost place. When one of the two buttons is pressed, the dialog hides, consumes the first token and donates a new token through the matching arc. As a result, either the user name is copied into the net, or the agent terminates.

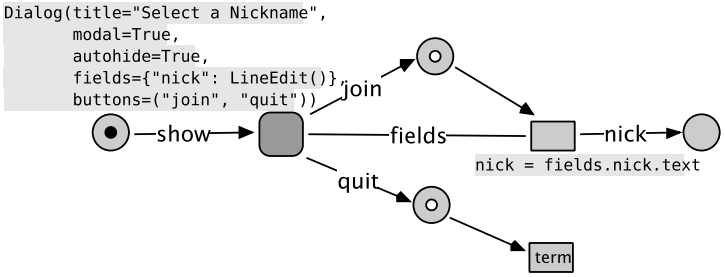

Fig. 2. A GUI gadget directly embedded in a net, interacting with a local user

Coarsening. The main mechanism of modularization in JAVA NETS is the nets within nets approach, since places can contain net objects. Contained nets can interact, through their own internal transitions, with outer transitions connected to their place, using synchronous channels. However, this approach introduces a sharp partitionment of the code into layers, resulting in a break of continuity.

PyTri favors a flat structure where all elements are on one universal layer, and other mechanisms for partitioning are used instead. Some control flows need to be executed in multiple independent instantiations, for example protocols that control interactions between agents. Instead of instantiating protocol nets as is done in MULAN, such code can be modeled through a new class of place. Parallel places transparently manage all bookkeeping necessary to flatten the behavior into a single structure, where tokens move in independent streams, separating the structure into virtual instances. Such places can also give an informational summary of all independent streams to the developer, during execution. This mechanism can be enhanced to allow piecing together net structures that look like AUML diagrams.

Coarsening and refinement of nets is a very powerful mechanism to make big structures controllable. By allowing to collapse and expand collections of elements forming a component, one can choose the optimal level of detail of the representation of the code. This can be used to modularize the code and the resulting segments can be coarsened to look just like normal elements. An interface for such a segment can be defined by specifying which of the contained elements are accessible through certain aliases. Depending on whether only places, transitions, or both are accessible, segments are represented as such an element, in the mixed case as a gadget.

Tool Support. An integral part of PyTri is having tool support for the language, namely a Monipulator that allows creation, manipulation, monitoring and debugging of net code. In order to allow developers to separate big nets into manageable chunks, they can be partitioned into fragments, which are joined into a flat whole net, maintaining the original semantics.

The preliminary maxim of PyTri, which arose due to the global interpreter lock of Python, is that every agent is its own process. There is no global simulator process that contains, manages and controls agents. Memory is not shared between agents, which have to rely on interprocess-interaction, facilitated by a process present on every machine of a distributed environment.

4 Conclusions

Programing complex systems remains a challenge. This paper provides a vision in form of a new graphical programming language to address this problem. PyTri overcomes some drawbacks of the successful extension of Java in the form of reference nets with agent concepts from MULAN. It does so by amending a different language, namely Python, and adding several new or adapted constructs and concepts to the language with respect to the graphical underpinning. While it is

strongly influenced by Petri nets, its semantics is adapted to the requirements of programming and explores a lot of different possible programming mechanisms, in order to investigate the realm of possibility. This allows for powerful expression of behavior through new constructs and concepts. PyTri's concepts, the prototypical implementation, written in Stackless Python, using PyQt, and the whole background of the language has been intensively discussed in [10]. Currently the main features are: Python as the inscription language, a powerful semantics with concepts and constructs to express important agent features, and an available prototype for some of the central concepts.

Our current direction of research is now to provide a proper Petri net semantics for the current concepts and constructs. Concepts and experiences from reference nets, Renew and Mulan/Capa will give further directions of development beside the direct evaluation of the current PyTri tool set. The main goal is to keep and extend PyTri as a powerful modeling and programming language while trying to regain the analytical power of Petri nets, which is currently lost. In order to truly seize the power of a language which directly expresses concurrency, a different implementation base for the execution core and the inscription language has to be considered, possibly based on PyPy or C++. Besides PyTri as a language, the tools around PyTri will also be subject to several different investigations. In the future, a series of prototypes, no longer based on Stackless, which gradually implement more and more concepts of the language, will be created. Here concrete, at the beginning rather artificial applications will further help to find the right modeling and programming concepts.

References

1. Apple Developer Connection – Quartz Composer. http://developer.apple.com/graphicsimaging/quartzcomposer/ (2009)
2. Cabac, L.: Modeling Agent Interaction Protocols with AUML Diagrams and Petri Nets. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg (Dec 2003)
3. Cabac, L.: Modeling Petri Net-Based Multi-Agent Applications. Logos Verlag, Berlin (2010)
4. Cabac, L., Dörges, T., Rölke, H.: A monitoring toolset for Petri net-based agent-oriented software engineering. In: Valk, R., van Hee, K.M. (eds.) 29th Petri net conference, Xi'an, China. LNCS, vol. 5062, pp. 399–408. Springer-Verlag (Jun 2008)
5. Jensen, K.: Coloured Petri Nets : Basic Concepts, Analysis Methods and Practical Use. Volume 1 (Monographs in TCS. An EATCS Series). Springer-Verlag (1992)
6. Johnson, G., Jennings, R.: LabVIEW graphical programming. McGraw-Hill Professional (2006)
7. Kummer, O.: Referenznetze. Logos Verlag, Berlin (2002)
8. Repenning, A., Ioannidou, A.: Agent-based end-user development. Commun. ACM 47(9), 43–46 (2004)
9. Russell: Workflow control-flow patterns: A revised view. http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf (2006)
10. Simon, J.: Design of a Python Based Visual Agent Programming Language and its Prototypical Implementation. Bachelor thesis, University of Hamburg, Department of Informatics (2009)