Modeling Reusable Concurrent Passive Entity Objects in Colored Petri Nets

Rowland Pitts and Hassan Gomaa

George Mason University, Fairfax, Virginia, USA
{rpitts,hgomaa}@gmu.edu

Abstract. Concurrent software systems are growing increasingly large and complex; the risks associated with poor design and architectural choices are increasing as well. Building executable prototypes can help identify problems early and Colored Petri Nets are well suited to this purpose. This paper presents an approach to modeling reusable thread-safe passive entity objects in Colored Petri Nets, including public, private and static members, plus encapsulation and object composition.

Keywords: colored Petri nets, concurrency, rapid prototyping, passive entity object.

1 Introduction

Concurrent software systems are growing increasingly large and complex. Consequently, the risks associated with poor design and architectural choices are increasing as well. Assembling executable models can help to identify problems early, and Colored Petri Nets (CPN) [9] are well suited for building executable concurrent software models; additionally, the language primitives facilitate the modeling of reusable design pattern templates [7], as well as the passive entity objects they interact with.

In spite of the fact that failure is increasingly expensive [1], often little consideration is given to system performance or reliability until a project is already implemented; unplanned behavioral analysis is typically inefficient, unreliable and difficult to repeat [12].

CPNs routinely depict concurrent software systems as tokens moving through a series of operations (transitions), sequentially or navigating control structures, analogous to dynamic flow charts. This paper introduces an approach to modeling thread-safe objects, with an emphasis on object-oriented properties, such as information hiding, providing a public interface of operations, and reusability using CPN Tools [5].

This paper is organized as follows: Section 2 discusses related work, Section 3 introduces the modeling approach and Section 4 provides validation. Section 5 discusses conclusions and future work.

2 Related Work

There is much literature devoted to the analysis of concurrent software with CPNs, and some related to object modeling.

Bauskar and Mikolajczak modeled objects using CPN's hierarchical capabilities [3]. Jensen and Kristensen have examined reusability using CPNs hierarchical capabilities [9]. Costa and Gomes propose module replication, composition and defining interfaces [4]. Barros and Gomes discuss transitions as functions with input parameters and also the creation and destruction of objects [2]. Pettit, Fant and Gomaa have modeled behavioral design patterns and communication templates, including threads-of-control [7, 12, 11]. Lakos introduces Object Petri Nets, which incorporate inheritance, polymorphism, dynamic binding, and include a single class hierarchy of both token and subnet types [10]. The Reference Net Workshop supports object references as tokens [13].

This paper focuses on combining a number of object-oriented properties while modeling concurrent objects, such as information hiding, providing a public interface of operations, static variables and operations, and reusability, as well as modeling threads-of-control by which a client can animate passive entity objects as needed.

3 Object Modeling in CPN

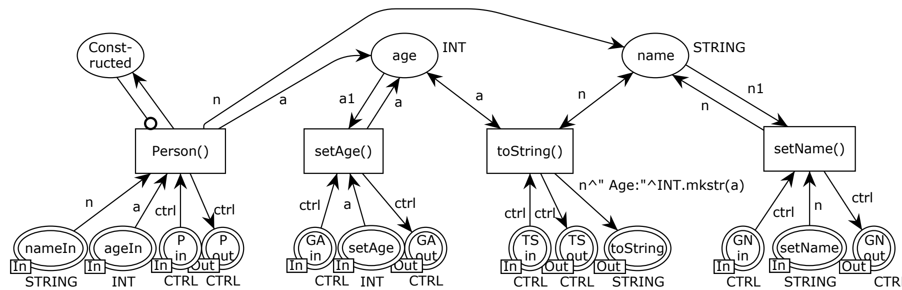

Fig. 1. Person class definition in CPN.

3.1 Design Conventions

CPNs are not inherently object-oriented; however, the language primitives allow for almost infinite flexibility. To the extent that visual structure aids in conveying a designer's intent, the following conventions, illustrated in Figure 1, are utilized for object modeling. The behavior otherwise modeled in Figure 1 is discussed in more detail in the next subsection.

Input and Output Parameters are depicted across the bottom of their class diagrams, grouped by operation, and indicated by a double-line, as opposed to a single line. This includes threads-of-control, which determine the sequence in which modeled operations execute. Placing tokens into the input places, and retrieving tokens from the output places, is the means by which clients communicate with objects.

Operations comprising a class' public interface are represented as transitions just above, and connected by arcs to, their respective inputs and outputs. Modeling operations as transitions works well for multiple reasons: transitions perform conversions, and CPN Tools' hierarchical capabilities facilitate the creation of reusable objects that effectively enforce communication through the defined public interface and otherwise prevent access to an object's non-public members.

Instance Variables are depicted as places in the space above the public interface operations, or as objects as described herein, and are maintained by the public operations or by other internal functions.

3.2 A Simple Class Example

Figure 1 is the class definition for a simple Person class. Two places near the top represent instance variables for age and name. Four public operations are provided: `Person()`, `setAge()`, `toString()` and `setName()`, and each is represented by a transition. Their various inputs and outputs are represented by places across the bottom.

Concurrency: No two operations can simultaneously access an object's values. No operation can execute until the constructor has been initially executed. Furthermore, the constructor cannot re-execute after the object is created.

Encapsulation: When used, a Person object's data elements and functionality are encapsulated within the object, providing the client with only indirect access through the defined public operations. An example Employee object is depicted in the uppermost region of Figure 2.

Reusability: Any number of Person objects may be used within a CPN.

3.3 A More Complex Example

Figure 2 represents an Employee class definition, which features a composed object (`Person1`), a static variable (`employeeCount`) and associated static accessor method (`getCount()`), and a meta-variable (`lock`) used for synchronization. Employee also includes a constructor (`Employee()`) and a `toString()` operation.

Given the relative complexity of this example, representing an operation with a single transition is insufficient. Treating these as atomic actions would result in the thread-of-control being released to the client prematurely, and potentially cause concurrency issues. Therefore, an inbound transition fires to initiate the behavior sequence, and a return transition fires when the process is complete, releasing the thread-of-control and return values at the appropriate time.

For simplicity, a minimal number of operations have been modeled; however, more could easily be added. For example, `setAge()` and `setName()` could be added, and connected to the otherwise unused equivalents in `Person1`.

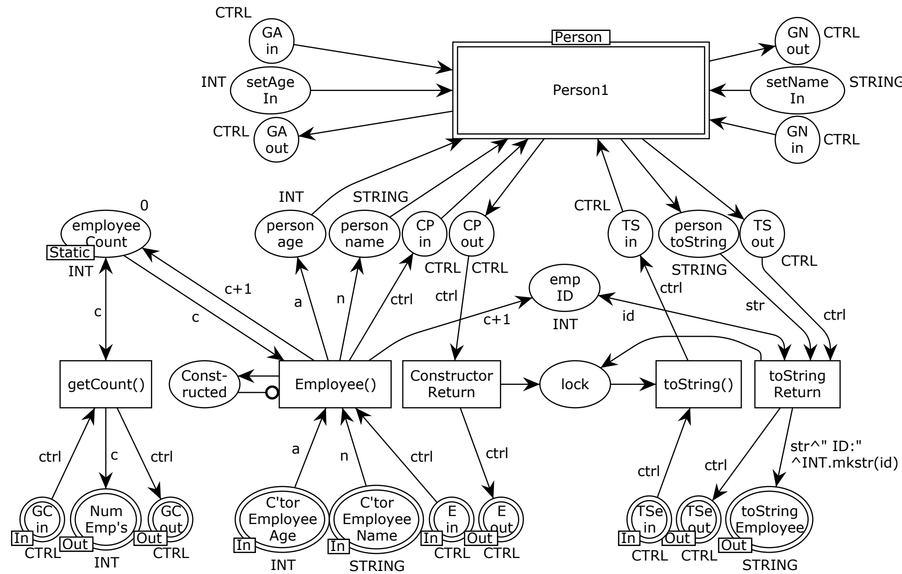

Fig. 2. Employee class definition, with composed Person object and static members.

Concurrency: The Employee class employs a more explicit locking mechanism. When the constructor executes, a token is moved to the `lock` place. To ensure mutually exclusive access, each instance-method must acquire the lock before executing and return it when finished [8]. Therefore, no two can execute simultaneously (given the scope of this short paper, only one such method is depicted, but any additional methods would acquire and release the lock token in the same way). Non-static methods cannot execute until the constructor has been invoked to create the object, and the constructor cannot re-execute once the object is created.

Encapsulation: An Employee object's data elements, including a Person object, and functionality are encapsulated. Employee provides only indirect access to itself through the defined public operations.

Reusability: Any number of Employee objects may be used within a CPN; additionally, each Employee object also re-uses a Person object.

Static Behavior: The `employeeCount` place is effectively made static by defining it as a fusion place, facilitated by CPN Tools. Its initial marking is zero (simulating an initialized value), and is incremented each time an instance of Employee's constructor is invoked. The value in the fusion place is shared by all instances of Employee; therefore, invoking the `getCount()` method in any instance of Employee will return the same value.

4 Validation

The limited length of this short paper permits only a brief description of the validation carried out; however, tests were conducted to determine that each modeled object's operations execute correctly and that the synchronization considerations ensure that there is no detrimental conflict for shared data. A unit testing approach was employed, because it offers "the most effective means to test individual software components for boundary value behavior" [6]. Figure 3 is a depiction of one such test scenario

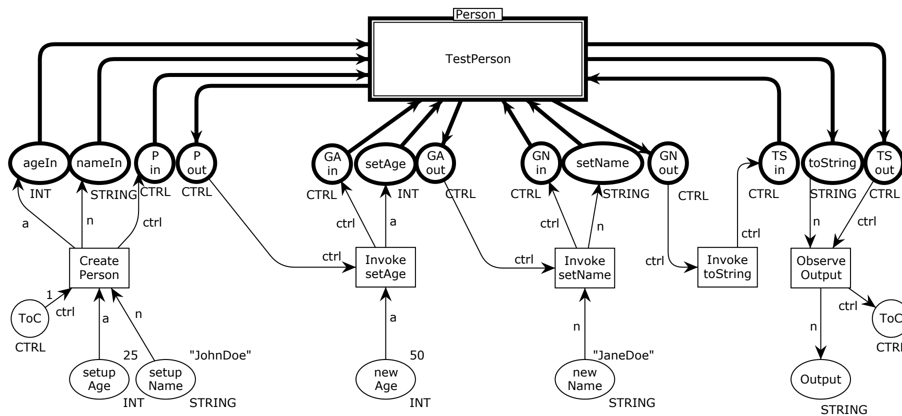

Fig. 3. Unit test of a Person object.

For clarity, the object under test, including the input and output places associated with its public operations, is depicted with bolder lines. The elements otherwise associated with the testing operations are depicted normally.

Test Scenario: From left to right in Figure 3, a Person object is created, after which the `setAge()` and `setName()` operations are invoked. Finally, the `toString()` operation is invoked in order to observe the expected output.

5 Conclusions and Future Work

Objects can be effectively modeled with CPNs, as shown in the examples above. The unit tests conducted confirm that they perform as expected. Modeling single- and multithreaded active objects is the logical next direction related to this short paper. Modeling inheritance would pose an interesting challenge as well.

This work is part of a larger project to model concurrent distributed applications and middleware. The ultimate goal of the overarching research effort is to provide a suite of executable architectural components and communications templates for a variety of software design patterns.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2016)
2. Barros, J.P., Gomes, L.: On the Use of Coloured Petri Nets for Object-Oriented Design, pp. 117–136. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
3. Bauskar, B.E., Mikolajczak, B.: Abstract node method for integration of object oriented design with colored Petri nets. In: Third International Conference on Information Technology: New Generations (ITNG'06). pp. 680–687 (April 2006)
4. Costa, A., Gomes, L.: Module composition within Petri nets model-based development. In: 2007 International Symposium on Industrial Embedded Systems. pp. 316–319 (July 2007)
5. CPN Tools website (May 2017), `http://cpntools.org`
6. Ellims, M., Bridges, J., Ince, D.C.: Unit testing in practice. In: 15th International Symposium on Software Reliability Engineering. pp. 3–13 (Nov 2004)
7. Fant, J.S., Gomaa, H., Pettit, R.G.: A comparison of executable model based approaches for embedded systems. In: 2012 Second International Workshop on Software Engineering for Embedded Systems (SEES). pp. 16–22 (June 2012)
8. Gomaa, H.: Real-Time Software Design for Embedded Systems. Cambridge University Press (2016)
9. Jensen, K., Kristensen, L.M.: Colored Petri nets: A graphical language for formal modeling and validation of concurrent systems. Commun. ACM 58(6), 61–70 (May 2015)
10. Lakos, C.: Object Oriented Modelling with Object Petri Nets, pp. 1–37. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
11. Pettit, R.G., Gomaa, H., Fant, J.S.: Modeling and prototyping of concurrent software architectural designs with colored Petri nets. In: International Workshop on Petri Nets and Software Engineering. pp. 67–79 (2009)
12. Pettit, R.G., Gomaa, H.: Modeling behavioral design patterns of concurrent objects. In: Proceedings of the 28th International Conference on Software Engineering. pp. 202–211. ICSE '06, ACM, New York, NY, USA (2006)
13. The reference net workshop website (May 2017), `http://www.renew.de`