Chapter 9
An Introduction to Reflective Petri Nets

Lorenzo Capra
Università degli Studi di Milano, Italy

Walter Cazzola
Università degli Studi di Milano, Italy

ABSTRACT

Most discrete-event systems are subject to evolution during lifecycle. Evolution often implies the development of new features, and their integration in deployed systems. Taking evolution into account since the design phase therefore is mandatory. A common approach consists of hard-coding the foreseeable evolutions at the design level. Neglecting the obvious difficulties of this approach, we also get system's design polluted by details not concerning functionality, which hamper analysis, reuse and maintenance. Petri Nets, as a central formalism for discrete-event systems, are not exempt from pollution when facing evolution. Embedding evolution in Petri nets requires expertise, other than early knowledge of evolution. The complexity of resulting models is likely to affect the consolidated analysis algorithms for Petri nets. We introduce Reflective Petri nets, a formalism for dynamic discrete-event systems. Based on a reflective layout, in which functional aspects are separated from evolution, this model preserves the description effectiveness and the analysis capabilities of Petri nets. Reflective Petri nets are provided with timed state-transition semantics.

INTRODUCTION

Evolution is becoming a very hot topic in discrete-event system engineering. Most systems are subject to evolution during lifecycle. Think e.g. of mobile ad-hoc networks, adaptable software, business processes, and so on. Such systems need to be updated, or extended with new features, during lifecycle. Evolution can often imply a complete system redesign, the development of new features and their integration in deployed systems.

It is widely recognized that taking evolution into account since the system design phase should be considered mandatory, not only a good practice. The design of dynamic/adaptable discrete-event systems calls for adequate modeling formalisms and tools.

DOI: 10.4018/978-1-60566-774-4.ch009

Unfortunately, the known well-established formalisms for discrete-event systems lack features for naturally expressing possible run-time changes to system's structure.

System's evolution is almost always emulated by directly enriching original design information with aspects concerning possible evolutions. This approach has several drawbacks:

- all possible evolutions are not always foreseeable;
- functional design is polluted by details related to evolutionary design: formal models turn out to be confused and ambiguous since they do not represent a snapshot of the current system only;
- evolution is not really modeled, it is specified as a part of the behavior of the whole system, rather than an extension that *could* be used in different contexts;
- pollution hinders system's maintenance and reduces possibility of reuse.

Petri nets, for their static layout, suffer from these drawbacks as well when used to model adaptable discrete-event systems. The common modeling approach consists of merging the Petri net specifying the base structure of a dynamic system with information on its foreseeable evolutions. A similar approach pollutes the Petri net model with details not pertinent to the system's current configuration. Pollution not only makes Petri net models complex, hard to read and to manage, it also affects the powerful analysis techniques/tools that classical Petri nets are provided with.

System evolution is an aspect orthogonal to system behavior, that crosscuts both system deployment and design; hence it could be subject to separation of concerns (Hürsch & Videira Lopes, 1995), a concept traditionally developed in software engineering. Separating evolution from the rest of a system is worthwhile, because evolution is made independent of the evolving system and the above mentioned problems are overcome. Separation of concerns could be applied to a Petri net-based modeling approach as well. Design information (in our case, a Petri net modeling the system) will not be polluted by non pertinent details and will exclusively represent current system functionality without patches. This leads to simpler and cleaner models that can be analyzed without discriminating between what is and what could be system structure and behavior. Reflection (Maes, 1987) is one of the mechanisms that easily permits the separation of concerns.

Reflection is defined as the activity, both *introspection* and *intercession*, performed by an agent when doing computations about itself (Maes, 1987). A reflective system is layered in two or more levels (base-, meta-, meta-meta-level and so on) constituting a *reflective tower*; each layer is unaware of the above one(s). Base-level entities perform computations on the application domain entities whereas entities on the meta-level perform computations on the entities residing on the lower levels. Computational flow passes from a lower level (e.g., the base-level) to the adjacent level (e.g., the meta-level) by intercepting some events and specific computations (*shift-up action*) and backs when meta-computation has finished (*shift-down action*). All meta-computations are carried out on a representative of lower-level(s), called *reification*, which is kept *causally connected* to the original level. For details look at Cazzola, 1998.

Similarly to what is done in Cazzola, Ghoneim, & Saake, 2004, the meta-level can be programmed to evolve the base-level structure and behavior when necessary, without polluting it with extra information. In Capra & Cazzola, 2007 we apply the same idea to the Petri nets domain, defining a reflective Petri net model (hereafter referred to as Reflective Petri nets) that separates the Petri net describing a system from the high-level Petri net (Jensen & Rozenberg, 1991) that describes how this system evolves upon occurrence of some events/conditions. In this chapter we introduce Reflective Petri nets, and we propose a simple