Science of Computer Programming 83 (2014) 3–34

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Taming distributed system complexity through formal patterns

José Meseguer

Computer Science Department, University of Illinois at Urbana-Champaign, IL 61801, USA

HIGHLIGHTS

- Solutions to frequently occurring problems in distributed systems can be made generic and reusable as formal patterns.
- Such formal patterns can greatly reduce the complexities of designing, verifying, and implementing a system.
- Formal patterns can be made executable in rewriting logic and come with semantic applicability conditions and formal guarantees.
- The paper defines the semantics of formal patterns and illustrates their usefulness in various cyber-physical, medical, and security applications.

ARTICLE INFO

Article history:
Received 3 December 2012
Received in revised form 3 July 2013
Accepted 4 July 2013
Available online 24 July 2013

Keywords:
Software patterns
Distributed systems
Formal specification and verification
Rewriting logic
Maude

ABSTRACT

Many distributed systems are real-time, safety-critical systems with strong qualitative and quantitative formal requirements. They often need to be reflective and adaptive, and may be probabilistic in their algorithms and/or their operating environments. All this makes these systems quite *complex* and therefore hard to design, build and verify. To tame such system complexity, this paper proposes *formal patterns*, that is, formally specified solutions to frequently occurring distributed system problems that are generic, executable, and come with strong formal guarantees. The semantics of such patterns as theory transformations in rewriting logic is explained; and a representative collection of useful patterns is presented to ground all the key concepts and show their effectiveness.

1. Introduction

Many distributed systems are: (i) real-time; (ii) safety-critical, with strong qualitative and quantitative formal requirements; (iii) reflective and adaptive, to be able to operate in changing and potentially hostile environments; and (iv) possibly probabilistic in their algorithms and/or their operating environments.

Their distributed features, their adaptation needs, and their real-time and probabilistic aspects make such systems quite *complex* and therefore hard to design, build and verify. One important source of complexity, causing many unforeseen design errors, arises from ill-understood and hard-to-test *interactions* between their different *distributed components*. For real-time distributed systems this complexity is even higher, because such interactions must take place within specific time windows. All this system complexity makes system verification quite hard, yet unavoidable, since the *safety-critical* nature of many of these systems makes their verification essential.

All this means that methods to tame and greatly reduce *system complexity* are badly needed. System complexity has many aspects, including the complexity and associated cost of: (i) designing; (ii) verifying; (iii) implementing; and (iv) maintaining and evolving such systems.

E-mail address: meseguer@illinois.edu.

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.07.004

The main goal of this paper is to propose the use of *formal patterns* to drastically reduce all system complexity aspects. By a "formal pattern"[1] I mean a solution to a commonly occurring software problem that is:

1. As *generic*[2] as possible;
2. *Formally specified*, with precise applicability requirements;
3. *Executable*; and
4. Comes with *strong formal guarantees*.

A formal pattern can be applied to a *potentially infinite* set of concrete instances, where each such instance application is *correct by construction* and enjoys the formal guarantees of the pattern.

A key idea, also outlined in [51] for cyber-physical systems, is that distributed systems should be designed, verified, and built by *composing formal patterns* that are highly generic and reusable and come with strong formal guarantees. In this way, a large part of the verification effort can be done in an *up-front*, fully generic manner, so that it can be *amortized* across a potentially infinite number of instances.

As I will show through concrete examples, *drastic reductions in all aspects of system complexity*, including the formal verification aspect, can be achieved this way. The resulting system simplicity and component reusability should enable the design and implementation of high-quality, highly reliable distributed systems at a fraction of the cost required not using such patterns.

1.1. The need for a semantic framework

To develop *formal* patterns for distributed systems with features such as those mentioned above an appropriate *semantic framework* is needed, one supporting:

1. Concurrency and distributed objects;
2. Logical reflection;
3. Distributed object reflection and adaptation;
4. Executability;
5. Real time and probabilities; and
6. Formal verification methods and tools.

Supporting features (1)–(6) is a tall order. Yet, as the examples I will present show, all these features are necessary to give a formal semantics to patterns for distributed systems. I will use *rewriting logic* as a semantic framework supporting features (1)–(6), and will show through a substantial collection of examples its adequacy to specify and verify formal patterns of this nature.

1.2. Rewriting logic in a nutshell

Rewriting logic [36] is a flexible logical framework to specify concurrent systems, where a concurrent system is specified as a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ with:

- Σ a signature defining the *syntax* and *types* of the system;
- E a set of equations defining system's *states* as an *algebraic data type*; and
- R a set of *rewrite rules* of the form $t \to t'$, specifying the system's *local concurrent transitions*.

Rewriting logic deduction consists in applying rewriting rules R *concurrently*, *modulo* the equations E, so that in a system specified by a rewrite theory $\mathcal{R}$, concurrent computation *coincides* with logical deduction in $\mathcal{R}$.

The above remarks, plus the fact that it can naturally specify distributed object systems [37], show that rewriting logic supports feature (1) in the list of framework requirements of Section 1.1. Feature (2), *logical reflection* [20], is also key, as I further explain in Section 2, where I also show how various forms of distributed object reflection (feature (3)) are naturally supported. Under simple conditions, rewrite theories are executable (feature (4)) in rewrite engines such as Maude [19], CafeOBJ [29] and Elan [14]. Furthermore, the Real-Time Maude [48] and the PMaude [4] language extensions respectively

[1] Throughout this paper, "formal patterns" will be the formal counterpart of what are called *design patterns* in the informal descriptions of patterns, e.g., [30]. Therefore, they are computational solutions with a clear algorithmic meaning. In their informal descriptions this is exemplified by sample code in a conventional language such as, say, C++. All this is captured at the formal level by requirement (4) below, insisting that the formal specification of a (design) pattern should be *executable*.

[2] Throughout this paper I will always assume that a formal pattern is a *generic component*. For informal patterns this was not part of the original concept. However, the work of Meyer and Arnout [43] has shown a general way in which patterns can be turned into generic components.