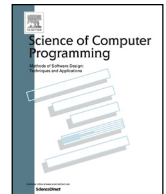
Science of Computer Programming 83 (2014) 3–34

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.

© 2013 Published by Elsevier B.V.

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 © 2013 Published by Elsevier B.V.
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.

support the specification of real-time and probabilistic distributed systems (feature (5)). Also, for verification purposes, Maude additionally provides several *model checkers* and *theorem proving* tools [19]. Likewise, Real-Time Maude provides a model checker [48]; and properties of probabilistic rewrite theories in PMaude can be model checked using the PVeStA tool [8], which allows parallelization of the statistical model checking process for better performance and scalability (feature (6)).

1.3. Paper organization

Since the word "pattern" can be used rather vaguely, the expression "formal pattern" may be infected by such vagueness, which would make all formal specification and verification claims highly questionable. This of course would be unacceptable. So the first order of business is for me to explain precisely what I mean by a "formal pattern" within the semantic framework of rewriting logic. This I do in Section 2.

But how useful is this notion in practice? The main goal of this paper is to try to convince you, my reader, that it is very useful to drastically simplify the design and verification of distributed systems. This means that I should provide some hard evidence showing that this is not a pie-in-the-sky idea. Fortunately, I have hard evidence in hand, based on a variety of formal patterns developed with other colleagues, which will help ground the ideas that I am proposing, including:

1. The *Command Shaper Pattern* for safe interoperation of medical devices proposed in [54], which is discussed in Section 3.
2. The *PALS* pattern for achieving virtual synchrony in a distributed real-time system originally proposed in [45], and formalized and proved correct in [39,40], which is explained in Section 4.
3. The *Cookies* pattern for protection against Denial of Service (DoS) attacks was proposed in [17] and is summarized in Section 5.1.
4. The *Adaptive Selective Verification (ASV)* DoS protection protocol was originally proposed in [34]. It was formalized and analyzed as a probabilistic rewrite theory in [9], and, as explained in Section 5.2, was specified in a modular way as a formal pattern in [6,27].
5. The *Server Replicator (SR)* pattern proposed in [27] can provision additional servers to decrease server load as explained in Section 5.3.
6. The *Composition of the ASV and SR* into the *ASV* + *SR* pattern proposed in [27] to maintain a uniform level of service in spite of an increasingly harsh DoS attack, which is explained in Section 6.1.
7. The *Multirate PALS* pattern proposed in [11], which achieves virtual synchrony in a multirate distributed real-time system by *composing* the *PALS* pattern described in Section 4 with two simpler patterns: the *k-Step Machine* pattern, and the *Input Adaptor* pattern, is explained in Section 6.2.

Since details can be found in the already published papers [54,40,34,9,27], I will not describe all these patterns in full detail. What I will instead do in Sections 3–6 is to convey the main ideas about each pattern in a somewhat impressionistic way, referring to the corresponding publications for additional details.

Indeed, the main goal of this paper is not to describe the above patterns in full detail, but to *explain what formal patterns are*, giving a precise semantics for them. A related goal is to explain why the formal guarantees associated to each pattern, together with their generic nature, can greatly simplify the design and verification of high-confidence distributed systems. The example patterns help ground the overall discussion.

I discuss related work and present some concluding remarks in Section 7.

2. Formal patterns as theory transformations

As already mentioned, what I intuitively mean by a "formal pattern" is a solution to a commonly occurring software problem that is: (i) as *generic* as possible; (ii) *formally specified*, with precise semantic requirements for its applicability; (iii) *executable*; and (iv) comes with *strong formal guarantees*. This, however, does not explain what a formal pattern is mathematically: this requires using a formal framework in which the *semantics* of such patterns can be made precise. This is exactly what I do in this section, using rewriting logic as the framework. I begin by proposing a taxonomy of formal patterns, and then proceed to make precise their associated semantics.

2.1. A taxonomy of formal patterns

Without claiming that the kinds of formal patterns below form an exhaustive taxonomy, they have been used extensively, have proved to solve many problems, and are general enough to cover many examples, including all the examples I will discuss:

1. **Parameterized Theories.** Example: The *Command Shaper* pattern, ensuring *stress–relax* safety of medical devices.
2. **Theory Transformations with Performance Parameters.** Example: The *PALS* pattern, ensuring *virtual synchrony* in real-time distributed systems.
3. **Reflective Meta-Object Patterns,** which can be:
   - **Onion-Skin Meta-Objects:** Examples: The *Cookies* and *ASV* meta-objects, ensuring different forms of *DoS protection*.
   - **Russian Dolls Meta-Objects:** Examples: The *Server Replicator* (SR) pattern for Cloud Computing, ensuring *uniform performance* under increasing client requests, and the ASV + SR pattern, composing ASV and SR.

An important point to make is that the just-listed kinds of patterns *are not mutually exclusive*. For example, the parameterized theories used in distributed system specifications often involve meta-objects, as it is indeed the case for the Command Shaper pattern. Conversely, meta-object patterns are often also parameterized by an input theory. That is, a given pattern may simultaneously fall under several of the above-mentioned categories. I describe each example pattern under the category that seems most salient given the pattern's characteristics, without implying that it is the only possible category under which it could be classified.

A second important point to make is that the above-listed kinds of patters are not an ad-hoc menagerie of pattern kinds. Instead, they are time-tested, *systematic* methods to provide general solutions to distributed system problems. Parameterized theories have a long history as a method to make formal specifications generic [15,32,31], and have for long been applied to programming languages in the form of parameterized modules [32], sometimes described as generic modules, polymorphic types, or templates. They have in particular been applied to rewriting-based distributed system specifications since [37,25,19]. Likewise, meta-objects have a long history as a modular method to mediate and adapt the interactions between distributed objects within the Actor framework [3,56], and have long been applied to many rewriting-based distributed system specifications since [21,41].

2.2. The semantics of formal patterns

Mathematically, what the patterns in the above taxonomy have in common is that they all are *theory transformations*. That is, a pattern, lets us call it $B[_,_]$, is *instantiated* by applying it to an *input theory* T in a class of theories satisfying precise semantic requirements, possibly with some *additional parameters* Γ. If T is an acceptable[3] input theory[4] and Γ the acceptable parameters, then $B[_,_]$ is a theory transformation:

$$(T,\Gamma) \mapsto B[T,\Gamma].$$

The fact that many input theories T can be used to instantiate the pattern $B[_,_]$ makes the pattern *generic*. Of course, $B[_,_]$ should *preserve executability*: if T is an executable rewrite theory, then $B[T,\Gamma]$ should also be executable. The idea that the instances $B[T,\Gamma]$ of the pattern $B[_,_]$ come with strong formal guarantees has a specific meaning for each pattern $B[_,_]$: each $B[_,_]$ is solving a different problem by meeting the specific formal requirements of that problem. This will become clear in examples.

Several aspects of an (informal) pattern discussed by the Gang of Four in [30] become now formalized by the theory transformation $B[_,_]$ that specifies the pattern. The so-called *context* and *applicability* of the pattern is now formalized by the semantic requirements that an input (T,Γ) to the pattern must satisfy. Likewise, the *intent* and *consequences* of the pattern come now with strong assume/guarantee formal properties: if the input (T,Γ) satisfies the pattern's semantic requirements, then $B[T,\Gamma]$ will satisfy specific formal properties associated to the pattern. Likewise, the *implementation* and *sample code* aspects are formalized by the *executable* formal specification of $B[_,_]$.

I explain below in more detail what such theory transformations look like for patterns of types (1)–(3) in the above taxonomy.

2.2.1. Parameterized theories and their pushout semantics

Parameterized theories are well known from *parameterized algebraic data types*, *parameterized formal specifications*, and *parametric polymorphism*. Parameterization has been studied as a very general mechanism available in many logics in the theory of institutions [31]. In particular, it is available in rewriting logic and supported by Maude [19], where parameterized rewrite theories define *parameterized concurrent systems*, so they are extremely useful.

Formally, a *parameterized theory* $B[P]$ with parameter P is a *theory inclusion* $P \overset{J}{\hookrightarrow} B$, where P is called the *parameter theory* and B the *body*. A simple example in the case of equational specifications (equational logic is a sublogic of rewriting logic) is the parameterized theory *Sorting*[*Toset*] specifying the algebraic data type of lists with a sorting function associated

[3] Acceptability means that the *domain* of the theory transformation $B[_,_]$, that is, the set of pairs (T,Γ) to which $B[_,_]$ can be applied producing a correct result must be explicitly specified, so that T and Γ satisfy specific *semantic requirements* which are part of the formal specification of the pattern. The strong formal guarantees associated with the pattern are *conditional* on (T,Γ) satisfying such semantic requirements and therefore can be viewed as an "assume/guarantee contract" honored by the pattern.

[4] In general, an n-ary transformation accepts a *tuple* $(T_1,\ldots,T_n)$ of input theories. For example, a coproduct of n theories is a transformation $(T_1,\ldots,T_n) \mapsto T_1 \oplus \cdots \oplus T_n$. Alternatively, if we have a tuple of theories, we can absorb the remaining $T_2,\ldots,T_n$ as part of the additional parameters Γ.

to a totally ordered set of list elements; that is, to a set with a total order relation $\leqslant$ satisfying the total order axioms specified in the theory *Toset*. Examples of parameterized theories defining parameterized concurrent systems will be given in subsequent sections.

The way such a parameterized theory $B[P]$ is used is by *instantiating* its formal parameter P with an acceptable instance. For the case of *Sorting*[*Toset*] an acceptable instance is precisely a set with a total order relation on it. If the semantic requirement of $\leqslant$ being a total order is satisfied for a concrete instance (an actual parameter), then the instantiation of *Sorting*[*Toset*] will work correctly. If, instead, the contract is violated, so that $\leqslant$ is not a total order for the given concrete instance, the instantiation is *incorrect* and no guarantees can be given about it due to the violation of the semantic requirements imposed by the theory *Toset*.

How can we describe a parameterized theory $B[P]$ as a theory transformation $(T,\Gamma) \mapsto B[T,\Gamma]$? Very naturally: the domain of such a theory transformation is the set of pairs (T,Γ) of the form (T,H) such that: (i) T is a theory in the given logic (in our case, rewriting logic), and (ii) H is a *theory interpretation*[5] $H : P \longrightarrow T$, that is, a mapping H of types in P to types in T, and of symbols in P to symbols (or, more generally, expressions) in T such that the axioms of P, say, Ax_P, when translated by the mapping H are logical consequences of T, that is, $T \models H(Ax_P)$. For any admissible argument (T,H), the transformation $(T,H) \mapsto B[T,H]$ associated to $B[P]$ is abbreviated to $(T,H) \mapsto B[H]$ (note that T can be retrieved as the co-domain of $H : P \longrightarrow T$), where $B[H]$ is defined by the pushout diagram:

$$\begin{array}{ccc} B[P] & \xrightarrow{H'} & B[H] \\ {\scriptstyle J}\uparrow & & \uparrow{\scriptstyle J'} \\ P & \xrightarrow[H]{} & T \end{array}$$

in the category *Th* whose objects are theories in the given logic (in our case rewrite theories), and whose morphisms are theory interpretations between such theories. The category *Th* of theories may include theories with initiality and freeness constraints (the "data theories" of [31]), and may furthermore support theory hierarchies, as done for the Maude language in [19], and explained for logics in general in [24].

Intuitively, we can think of the instantiation $B[H]$ as a way of "gluing together" the body B and the instance theory T, using the "gluing information" provided by H. Note that $B[P]$ may be instantiated in *several different ways* to a theory T, depending on the choice of H. For example, we can instantiate *Sorting*[*Toset*] to the theory *Nat* defining the natural numbers in two different ways: (i) using the theory interpretation $up : Toset \longrightarrow Nat$, which maps $\leqslant$ in *Toset* to $\leqslant$ in *Nat*, we get the instantiation *Sorting*[*up*], which sorts lists of natural numbers in ascending order; and (ii) using the theory interpretation $down : Toset \longrightarrow Nat$, which maps $x \leqslant y$ in *Toset* to $x \geqslant y$ in *Nat*, we get the instantiation *Sorting*[*down*], which sorts lists of natural numbers in descending order.

The formal guarantees associated to the pattern *Sorting*[*Toset*] include the assurance that the sorting function will perform correctly: given any list of elements, it will return a list involving the same elements, but now sorted according to the given total order. Such guarantees can only be ensured to hold if the semantic requirement that the theory interpretation $H : P \longrightarrow T$ is correct is met. They hold, for example, for *Sorting*[*up*] and *Sorting*[*down*]; but we could have instead defined a map, say *div*, mapping $\leqslant$ in *Toset* to the divisibility relation $_ \mid _$ in *Nat* where, by definition, $x \mid y \Leftrightarrow (\exists z) x \cdot z = y$. However, the mapping *div* is *incorrect*, because the total order axiom $(\forall x, y) x \leqslant y \vee y \leqslant x$ is *not* satisfied by *Nat* when translated by this mapping as $(\forall x, y)(x \mid y) \vee (y \mid x)$. That is, *div* does *not* define a theory interpretation $div : Toset \longrightarrow Nat$, although it does define a theory interpretation $div : Poset \longrightarrow Nat$ for *Poset* the theory of *partially* ordered sets. Therefore, no guarantees can be given about the incorrect instantiation *Sorting*[*div*], which would sort numbers based on divisibility. In particular, there is no guarantee anymore that we would always get the same results for the function *sort* (so that *sort* is actually a *function*, and not just a relation, on lists). For example, depending on how the equations defining the *sort* function are applied, we could get $sort(6; 3; 2) = 2; 3; 6$, or we could get $sort(6; 3; 2) = 3; 2; 6$.

2.2.2. Theory transformations with performance parameters

For real-time systems, certain *timing and performance requirements* are essential for system correctness. This means that some formal patterns for *distributed real-time systems*, such as, for example, the PALS pattern that I discuss in Section 4, can be defined as theory transformations

$$(T,\Gamma) \mapsto B[T,\Gamma]$$

where T is the input theory, Γ are suitable *performance parameters*, and the domain of correct inputs (T,Γ) is precisely specified. The formal guarantees of $B[T,\Gamma]$ can be seen as a *contract*, where if the pair (T,Γ) is a correct input for $B[_,_]$, then $B[T,\Gamma]$ satisfies the contract and comes with specific formal guarantees. For example, as I will further explain in Section 4, in the case of the PALS pattern, (T,Γ) being a correct input means that T specifies a synchronous composition of abstract machines; and that the performance parameters Γ can be guaranteed to hold for a distributed, asynchronous

[5] Also called a *view* in [32,19], since H gives us a sound way of *viewing* the theory T as an instance of P.

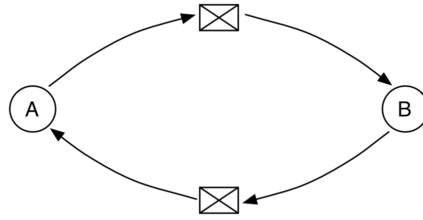


Fig. 1. Object system with messages in an idealized environment.

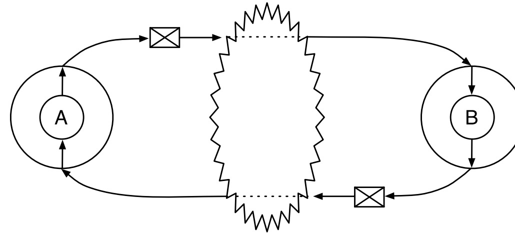


Fig. 2. Onion-skin wrapped system with messages in an hostile environment.

implementation of such a composition. If these requirements are met, the formal guarantee associated to the PALS pattern is that the synchronous composition of the input machines T and their asynchronous realization as the transformed $B[T, \Gamma]$ are *bisimilar*. For other real-time system patterns, the input theories T, the performance parameters Γ, and the formal guarantees ensured by the given pattern may of course be different.

2.2.3. *Reflective meta-object patterns*

Distributed systems need to *adapt* to changing, often hostile environments to ensure various safety, security, fault tolerance, real-time, and Quality of Service (QoS) requirements. This can be accomplished by *meta-object* patterns. Meta-objects are distributed objects that control/mediate/adapt the behavior of the underlying objects they encapsulate *without changing the behavior of such underlying objects in any way*. That is, a meta-object is a very modular, distributed component that can be composed with some existing distributed objects without changing their basic functionality and *mediates* inter-object communication in useful ways to achieve important system-level goals. In this way meta-objects allow a modular separation of concerns and can achieve high levels of reusability and adaptation.

Distributed meta-objects can be classified as:

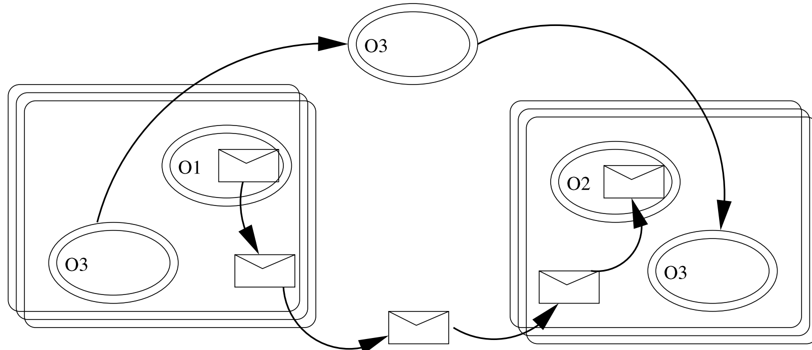
- **Onion-Skin** meta-objects, which wrap a *single object*. Examples: Cookies and ASV.
- **Russian Dolls** meta-objects, which wrap a *configuration*, that is, a collection of (possibly nested) objects. Example: SR.

Onion-skin meta-objects were proposed in [3], and were subsequently formalized in rewriting logic in [21]. As illustrated in Figs. 1–2, their purpose is to "wrap" existing objects, to avoid communication difficulties due to an hostile environment or to meet other system requirements. For example, meta-objects for encryption, resp. for fault-tolerant communication, may allow objects to communicate in an insecure environment, resp. a faulty one, as if no such problems existed.

The description of such meta-objects as "onion skin" emphasizes the fact that a wrapped object behaves *just as any other object*, in the sense that it continues to send and receive messages in the usual manner. This means that it is possible to *layer* several such wrappers on top of each other, like the layers of an onion. For example, a system may be protected against certain security threats *and* against communication faults by first wrapping it with a fault-tolerant meta-object, and then wrapping it again with a suitable encryption meta-object.

These ideas were later generalized to the so-called "Russian dolls" model [41], also formalized in rewriting logic, where a meta-object can mediate/coordinate/control not just a single object, but a potentially nested collection of objects; and where object-based reflection can be combined with logical reflection (see Section 2.4) for greater adaptability. The name "Russian dolls" refers to the famous Russian folk art dolls, which contain smaller dolls inside, and so on. But unlike the usual Russian dolls, a meta-object may contain not one but *several* immediate subobjects inside, each possibly containing several others in a nested way. The Russians dolls model contains the onion-skin model [3], the TLAM model [56], and the Mobile Maude language [23] as special cases.

For example, the figure below describes a nested configuration of objects and meta-objects in Mobile Maude, where objects can reside and execute inside rectangular meta-objects called "environments," each associated with a particular location, and can: (i) communicate with other objects in the same or in different environments; and (ii) move from one environment to another.



What is common to Russian doll meta-objects and to the simpler onion-skin meta-objects is that their semantics is characterized at the rewriting logic level by so-called *boundary-crossing* rewrite rules. For example, in the above figure, the fact that messages and objects are going in and out of given environments is governed by meta-object rewrite rules at the environment level that move objects and messages in and out of an environment. Likewise, since in Mobile Maude individual objects are also wrapped by an onion-skin wrapper, messages addressed to the inner object have to go inside the wrapper, and messages addressed to other objects have to go outside the wrapper, which is again accomplished by the wrapper's rewrite rules.

All this is in sharp contrast with an idealized system of objects and messages such as that in Fig. 1, where the distributed state is a *flat configuration*, that is, a non-hierarchical multiset or "soup," of objects and messages. In rewriting logic this is modeled by declaring a sort *Configuration* with a binary multiset union operator, usually denoted with empty infix syntax (juxtaposition), and having the sorts *Object*, of objects, and *Msg*, of messages, as *subsorts* [37,19]. For example, a flat configuration consisting of objects O_1, O_2, O_3, two copies of message M, and one copy of message M' is modeled as the flat multiset expression

$$O_1 \ O_2 \ O_3 \ M \ M \ M'$$

where the order of objects and messages in the "soup" is of course immaterial. Objects themselves are modeled as records of the form:

$$[Id, a_1 : v_1, \ldots, a_n : v_n]$$

where Id is the object's *name* or identity, the $a_1, \ldots, a_n$ are the object's *attributes*, and the $v_1, \ldots, v_n$ are the *values* that those attributes have in the current state of the object. In a distributed system whose states are flat configuration, rewrite rules describe communication events that can concurrently change some part of the soup. For example, an asynchronous reception of a message $(Id \leftarrow P)$ addressed to Id with payload P by object Id in state $atts$ and changing to state $atts'$ can be specified by a rewrite rule of the general form:

$$(Id \leftarrow P)[Id, atts] \longrightarrow \big[Id, atts'\big].$$

How can we pass from flat configurations to the nested configurations exhibited by meta-objects? That is, how can meta-objects be modeled? Very easily: as objects having an *internal configuration* as one of their attributes. For example, a meta-object named, say, Id may contain the above configuration objects O_1, O_2, O_3, two copies of message M, and one copy of message M' as part of its state in an attribute called *conf*, and possibly other attributes *atts*, as follows:

$$\big[Id, conf : O_1 \ O_2 \ O_3 \ M \ M \ M', atts\big].$$

Meta-object-specific boundary crossing rules then govern how messages, and possibly objects, move in and out of their enclosing meta-objects. For example, we may have a conditional rule to move messages out of a meta-object under certain conditions (for example, that the message is addressed to an object outside the inner configuration) having the general form:

$$[Id, conf : C \ M, atts] \longrightarrow \big[Id, conf : C, atts'\big] \ M \ \textit{if cond}$$

where C is a variable of sort *Configuration*, which stands for the remaining messages and (possibly nested) objects in the meta-object's inner configuration. Note that, once we allow nested configurations, there is nothing to prevent an inner configuration from containing other objects with inner configurations, and so on; which is the whole idea of the Russian dolls model. Of course, in specific instances such nesting of objects will have specific limits.

We are now ready to describe reflective meta-object patterns as theory transformations. They are transformations of the form

$$(T, \Gamma) \mapsto B[T, \Gamma],$$

where the rewrite theory T specifies the original distributed system as a collection of object classes satisfying specific semantic requirements[6] and having associated rewrite rules, and $B[T, \Gamma]$ extends T with appropriate *meta-objects* of the form:

- (Onion-Skin) $[Id, conf : O\ MC, atts]$, with Id a name, O the inner object being wrapped (often also having the same name Id), MC a variable of sort *MsgConfiguration*, specifying configurations whose only members are messages (that is, no other objects except for O are allowed, but extra messages to and from the object can reside in MC), and *atts* wrapper-specific attributes.
- (Russian Dolls) $[Id, conf : C, atts]$, with Id a name, C the (possibly nested) object configuration being wrapped, and *atts* wrapper-specific attributes.

Therefore, a meta-object is an object that encapsulates within its state either a single object O and messages to and from it (onion-skin), or, more generally, an entire configuration C of objects and messages (Russian dolls).

The meta-object *rewrite rules* typically describe boundary-crossing events and are *completely generic*, so as to apply to the widest possible variety of underlying objects under minimal assumptions. This makes meta-objects *completely modular*: their wrapped objects are, so to speak, *blissfully ignorant* of their being wrapped and behave exactly as before, so that their code does not have to be changed in any way.

For example, we can add onion-skin meta-objects for clients and servers in a client–server system, to provide some useful functionality such as, say, encryption, fault tolerance, or cookies for DoS protection; but such meta-objects do not depend on any particular details of the client–server system they are applied to, except for the general requirement that the system has a client–server architecture.

2.3. Composed patterns

Given the modular nature of formal patterns, composing them by nested application and by other theory operations is easy and entirely natural. It can be a very useful way to design and build modular, high-quality distributed systems that are much easier to understand and reason about than complex, monolithic systems.

For example, if the semantic requirements for each individual pattern application are met, we may: (i) first build distributed systems $B_1[T_1, \Gamma_1]$ and $B_2[T_2, \Gamma_2]$ out of smaller components T_1 and T_2 by respectively applying formal patterns B_1 and B_2; (ii) put the resulting systems together as a theory coproduct $B_1[T_1, \Gamma_1] \oplus B_2[T_2, \Gamma_2]$ (roughly a disjoint union, but for hierarchical theories as described in [24], they can still *share* subcomponents); and (iii) apply a third pattern B_3 to $B_1[T_1, \Gamma_1] \oplus B_2[T_2, \Gamma_2]$ to get the richer composed system

$$B_3\big[B_1[T_1, \Gamma_1] \oplus B_2[T_2, \Gamma_2], \Gamma_3\big].$$

That is, we can view patterns, and other theory constructions such as theory unions or theory coproducts, as *theory operations*, and can then combine them by means of *theory expressions* to obtain a powerful *module algebra* the same way that we combine simpler expressions in an algebraic calculus.

The number of such theory composition expressions using a given repertory of basic patterns and other theory-building operations is typically infinite, and not all such theory compositions may be equally useful in practice. There may however be *specific theory expressions* that *solve some commonly occurring problem* in a generic way and are very useful in practice. I call such theory expressions *composed patterns*. They play a role analogous to that of *derived operations* in algebra. For example, we can define exponentiation to the power n as the derived operation defined in terms of the basic multiplication operation $_ \times _$ by the equation

$$y^n = y \times \overset{n}{\cdots} \times y.$$

In like manner, if the theory composition just described above is useful not just for very specific T_1 and T_2 but much more generally, we can abstract it as a composed pattern of the form:

$$B_3\big[B_1[_, _] \oplus B_2[_, _], _\big].$$

I illustrate this idea in Section 6 with two useful examples of composed patterns:

- The $ASV + SR$ pattern, which achieves stable availability under DoS attacks by combining the *ASV* and *SR* patterns, and
- The *Multirate PALS* pattern, that achieves virtual synchrony in a Distributed Real-Time System (DRTS) with components running at different rates by combining the *PALS* pattern with the two simpler *k-Step Machine* and *Input Adaptor* patterns.

[6] For example, we will see later some meta-object patterns that require the theory T to describe a distributed object system having a *client–server* architecture.

As the discussion of these two patterns will show, composed patterns can often be *more than the sum of their parts*, in the sense that the behavior emerging from the composition may meet system goals and come with guarantees that none of the more basic patterns used in the composition could achieve by itself.

2.4. The importance of logical reflection

A theory T in a logic is a *meta-level* entity. We can think *inside* T at the so-called *object level*, for example by proving theorems about T. But how can we reason *across* different theories inside the logic, for example to manipulate or transform T itself? In particular, how can we formalize a theory transformation $(T, \Gamma) \mapsto B[T, \Gamma]$ by axioms *inside* the logic? Such axioms should belong to a theory, but *which* theory, when even T itself is not fixed but is a parameter ranging over a typically infinite class of theories?

Logical reflection is a very important property of rewriting logic [20] which answers all these questions in a fully satisfactory manner. The key idea is that rewriting logic can faithfully represent its own theories and their deductions, because it has a finitely presented rewrite theory $\mathcal{U}$ that is *universal*, in the sense that for any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) we have the following equivalence:

$$\mathcal{R} \vdash t \to t' \quad \Leftrightarrow \quad \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \bar{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle,$$

where $\overline{\mathcal{R}}$ and $\bar{t}$ are terms representing $\mathcal{R}$ and t as data elements of $\mathcal{U}$, and $\langle _ , _ \rangle$ is a pairing operator also in $\mathcal{U}$. Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have:

$$\mathcal{R} \vdash t \to t' \quad \Leftrightarrow \quad \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \bar{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle \quad \Leftrightarrow \quad \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \bar{t} \rangle} \rangle \to \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \ldots.$$

In particular, the universal theory $\mathcal{U}$ has a type, called *Theory*, whose terms meta-represent rewrite theories. That is, a theory T is meta-represented by the term $\overline{T}$ of type *Theory*. Then we can meta-represent a theory transformation $(T, \Gamma) \mapsto B[T, \Gamma]$ as a function[7] $\overline{B} : \mathit{Theory} \times \mathit{Params} \longrightarrow \mathit{Theory} : (\overline{T}, \overline{\Gamma}) \mapsto \overline{B}[\overline{T}, \overline{\Gamma}]$, where *Params* is a data type meta-representing the additional parameters of the transformation. Furthermore, a meta-theorem of Bergstra and Tucker [12] ensures that if the theory transformation is effective — that is, computable — we can always axiomatize $\overline{B}$ by means of a *finite* set of confluent and terminating equations within rewriting logic in a finitary extension of the universal theory $\mathcal{U}$.

Note that, although rewriting logic is a first-order formalism, reflection endows it with powerful higher-order features to represent formal patterns *within* the logic. Indeed, there are strong similarities (for which I use the $\approx$ symbol) between logical reflection and polymorphism in typed λ-calculi. For example, we have a strong similarity $\mathit{Theory} \approx \mathit{Type}$, between the type of all theories in rewriting logic and the type of all types in some polymorphic calculi. Likewise, there is a strong similarity between the typings: $\overline{\mathcal{U}} : \mathit{Theory} \approx \mathit{Type} : \mathit{Type}$, but without the inconsistencies that $\mathit{Type} : \mathit{Type}$ can create in some λ-calculi. Using these strong similarities, a formal pattern $(T, \Gamma) \mapsto B[T, \Gamma]$ corresponds to a polymorphic function $\lambda(T, \Gamma).B[T, \Gamma]$. In Maude, these higher-order features are efficiently supported by the `META-LEVEL` module [19], and by Maude's extensible module algebra [25].

3. Example 1: A pattern for medical device safety

This example illustrates the case of a pattern for cyber-physical systems which is a parameterized module for medical device safety called the *Command Shaper* pattern [54]. It ensures strong safety properties for a wide range of instances of the pattern. For this and subsequent patterns I will explain the:

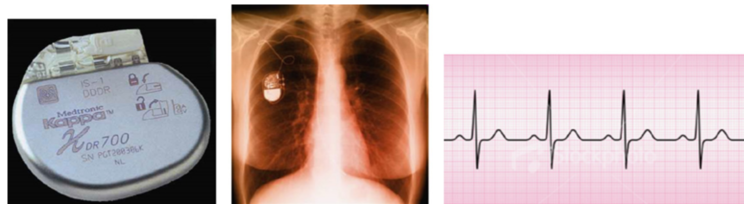
- *Problem*: what is the general problem that the pattern tries to solve?
- *Context*: what is the class of correct inputs to which the pattern can be applied?
- *Solution*: how is the pattern specified, and in which sense does it solve the original problem?
- *Guarantees*: what guarantees are provided by the pattern when correctly applied?
- *Advantages and Shortcomings*: what are the advantages, and what the possible limitations?

[7] Note that this function will in general be *partial*, since usually not all pairs $(\overline{T}, \overline{\Gamma})$ will be admissible arguments. In practice it is of course very important to precisely characterize the *domain* of the given transformation $\overline{B}$, that is, the set of its *correct inputs* $(\overline{T}, \overline{\Gamma})$. Such a domain can be characterized using membership equational logic [38], which is a sublogic of rewriting logic.

The problem addressed by the *Command Shaper* pattern can be illustrated by means of three examples and their associated safety requirements. The context of applicability is then precisely characterized through the parametric notion of an SR-safety theory. The solution is described as the Command Shaper pattern, which is a parameterized object-oriented module. The guarantees are the SR-safety properties ensured by the pattern. And the advantages and shortcomings are explained.

3.1. Problem: Illustrated by three medical devices and their safety requirements

Let us first consider an Implanted Cardiac Pacemaker as the one described in the figure below, where the device, its implantation in the patient's chest, and an electrocardiogram (ECG) are displayed.



Modern pacemakers do not stimulate the patient's heart at a fixed rate. Instead, they can sense and evaluate changes in patient motor activity through appropriate sensors and software and have a rate adaptation interface to adjust the patients heart rate during exercise. This makes the pacemaker more flexible and adaptable, but since now the rate is variable, this could become a safety hazard, unless the following two safety properties can be guaranteed:

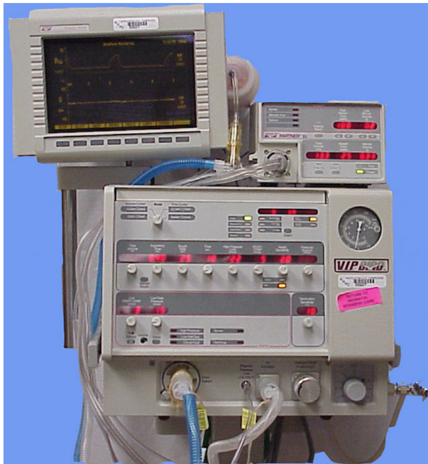
- The pacemaker must not pace too fast for *too long*, and should allow *sufficient resting time* between fast pacing periods;
- The pacemaker must not change the pacing rate *too drastically* over time.

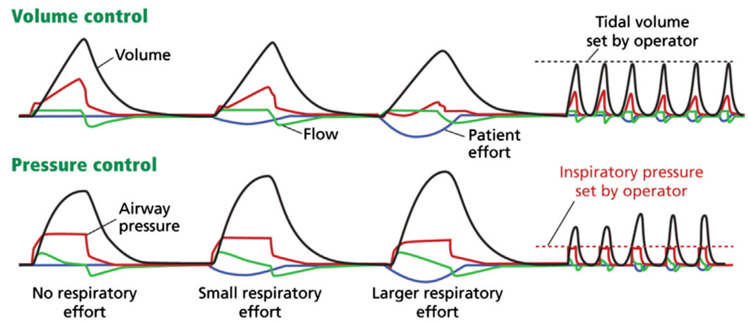
A second example belongs to the area of Patient Controlled Analgesia, where pain mitigation is provided by a *Morphine Infusion Pump* such as the one pictured below.

Such a pump has an interface to increase morphine injection rate (bolus dose) which can be operated by the patient. This allows the patient to obtain pain relief without the direct assistance of medical personnel, but this could become a safety hazard, unless the following two safety requirements can be guaranteed:

- Morphine injections must be administered with *sufficient time between doses*; and
- The total number of injections should not exceed a specified *maximum number of bolus* doses per hour.

The third example is a *Mechanical Ventilator* as the one pictured below.





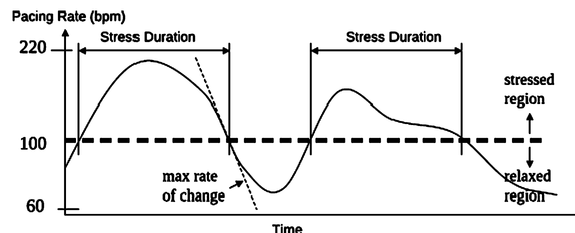
Such a device is used on a sedated patient, for example when performing a procedure under total anesthesia. It has a *pause* interface, because sometimes the ventilator needs to be temporarily stopped, for example to obtain an X-ray of the patient. This poses a serious potential safety hazard, namely, lack of oxygen in the patient's brain, unless:

- The ventilator is never paused for *too long*; and
- It is never paused *too often*.

3.2. Context: Stress–Relax Safety (SR-Safety)

These three examples illustrate a common theme about medical devices: for specific reasons patient's bodies may be placed under moderate stress temporarily, provided they are given sufficient time to recover. For each of the three medical devices we have considered, device states can be partitioned into two classes: *stressed* states and *relaxed* states. Stressed states are states in which the patient cannot stay for too long (heart pacing too fast, holding breath for too long, etc.), since otherwise permanent physical harm may result for the patient. Relaxed states are safe states that allow the patient to recover over time from a previous period of stressed states.

Safety in these circumstances can be characterized as *Stress–Relax Safety* (SR-Safety) [54], which imposes bounds on the *stress* and *relax* durations of device operation as well as on the rate at which relevant constants can change. Consider, for example, a pacemaker implanted on a patient for whom a safe minimal heart rate is 60 bpm when inactive. Furthermore, for this patient a critical heart rate is considered to be at 100 bpm, so that any heart rate above 100 bpm will be considered *stressed*, and any heart rate at or below 100 bpm will be considered *relaxed*. The key idea is that the pacemaker can pace at relaxed rates indefinitely without compromising patient safety, but should not pace at stressed rates for prolonged periods of time. Also, there should be enough distance between stressed periods. For example, if the pacemaker's pace drops below 100 bpm after a stress period, it should stay in the relaxed region for some time before allowing the pace to become stressed again. These safety constraints can be expressed as: (i) upper bounds on stress durations over time; (ii) minimum distance between stress periods; and (iii) a maximum pacing rate while on a stressed state and a minimum pacing rate for relaxed states, as shown in the following figure.



Furthermore, device states reflecting continuous physiological parameters should *change gradually*, to allow time for the patient to adapt to the effects. For example, it would be unsafe for the pacemaker when pacing at a rate of 100 bpm to immediately drop to 60 bpm over 1 second. This means that the rate of change in pacing speed (the absolute value of the "acceleration") should not exceed a prudent upper bound, as shown by the slope in the above figure.

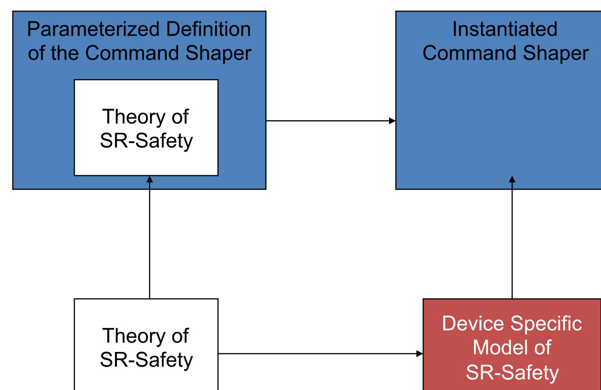
Of course, the same safety notions apply to the morphine infusion pump and the mechanical ventilator, except that the physical parameters involved (bolus doses, respirator pauses) are different from those of the pacemaker. In each case there will be similar safety bounds that should not be exceeded.

The general safety requirements applicable to all these devices can be axiomatized by means of an *SR-safety parameter theory* [54]. I do not describe this parameter theory in detail here. Instead, I summarize the main ideas that it captures. It applies to a general class of objects having an attribute that is a *value* of sort *Val* to be set by a message *set-val*. The problem, of course, is that the value to be set could be unsafe. To avoid this, the object will later be wrapped inside the *command shaper wrapper*. The safety of a value to be set depends on the domain-specific bounds of each device and on the history of previous settings. This is formalized by requiring that the set *Val* of values should be a *totally ordered set* with *minimum* and *maximum* values, and with an intermediate *critical* value, so that values less or equal to the *critical* one are *relaxed* values, and values greater than the *critical* one are *stressed*. The SR-safety of a *set-val* command trying to set the object to a certain value *v* does not only depend on *v* itself: it is *time-* and *history-dependent*. Assuming a given frequency with which new values are set, and assuming a *log* of previous setting commands, and certain bounds on the allowable rates of change in values, an *SR-safety envelope* can be defined, so that we can determine whether the new value to be set is *SR-safe* in the given state of the system. Being "SR-safe" means that setting the value in the current situation will not cause the patient to remain in a stressed situation for too long, will allow the patient to stay in a relaxed state long enough so as to recover from a prior stress, and will never change too drastically. If the given value is not SR-safe, then it should be replaced by an alternative SR-safe value that will keep the system within its SR-safety envelope.

Of course, all these notions are *parametric*. The choice of a specific set *Val* of values, and of all the required constants is domain-specific, and is precisely determined by a *theory interpretation*, so that, as usual for all parameterized modules, the class of admissible inputs for the *command shaper pattern* described below is exactly the class of all *theory interpretations* correctly instantiating the *SR-safety* parameter theory to a concrete domain of values satisfying all the axioms (as translated by such an interpretation) of the *SR-safety* theory. Note that such instantiations are not only domain-specific, so that the domain of values may differ quite a lot from, say, a pacemaker to a mechanical ventilator. They are also *patient-specific*, so that the choice of constants, such as, for example, the maximum, critical, and minimum values, will typically depend on the specific condition of the given patient and should be determined by a physician.

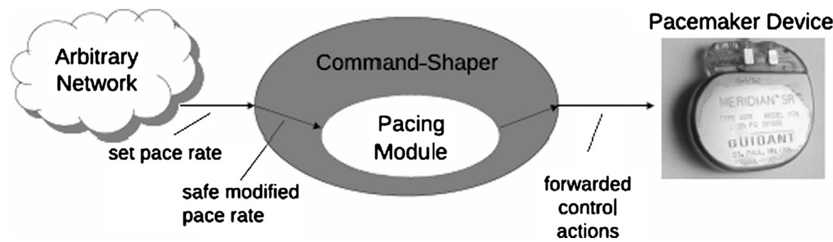
3.3. Solution: The command shaper pattern as a parameterized theory

The *command shaper pattern* is a *parameterized rewrite theory* in Real-Time Maude [54], whose instantiation for a given medical device used on a given patient is obtained as the following pushout diagram:



The key idea of the command shaper pattern is that each medical device is wrapped in a *command-shaper module*, a meta-object that monitors incoming device commands to ensure SR-safety [54]. The key function of the command-shaper module is to treat commands from external devices — for example, from the patient pushing a button for requesting a dose from the morphine infusion pump, or from the software adapting the pace of a pacemaker, or a pause command to a mechanical ventilator — not as actual commands to always be followed by the device, but as *suggestions* that should only be followed if they do not violate SR-safety.

The figure below shows the command shaper pattern applied as a meta-object wrapper around a pacing module in a cardiac pacemaker. If a command to modify the pacing rate is SR-safe, it is passed to the pacing module; but if it is detected to be SR-unsafe, the command-shaper can modify the command into a safe variant of it.



3.4. Guarantees

The command shaper pattern ensures that *SR-safety will never be violated*. In more detail, the following properties are satisfied by any correct input for the pattern:

- The state always remains within the allowed minimum and maximum values;
- The rate of change never exceeds the allowed bounds; and
- For the entire execution of the system, the states never remain stressed for too long, and enough recovery time is allowed between stress periods.

The last property is of course history-dependent; but since the command shaper pattern contains as one of its attributes a time-stamped *log* of prior stress-relaxed values, it can be characterized as an *invariant* in terms of the safety of such logs at any time.

3.5. Advantages and shortcomings

An important advantage of the command shaper pattern is that it can be applied to a wide range of medical devices to ensure their SR-safety under quite general conditions and is highly reusable. A significant limitation is that not all medical safety problems fall within the class handled by the command shaper. For example, the safety of a glucose–insulin pump depends on external context and sensor information, and cannot rely on any static notion of relaxed states.

4. Example 2: Achieving virtual synchrony with PALS

4.1. Problem: Achieving virtual synchrony in an asynchronous setting

Many *distributed real-time systems* (DRTS) are collections of components that communicate *asynchronously* through a network. Each component has a *local clock* and can change state and respond to inputs from the environment and from other components within *hard real-time bounds*. Furthermore, in many cases the DRTS as a whole should achieve *virtual synchrony*, so that the components all receive inputs from and send outputs to other components *at the right time*. In this way, the overall DRTS behaves *as if it were synchronous*. Such systems are common in application areas such as integrated modular avionics and distributed control in motor vehicles. They are often *safety-critical*, where, besides having to satisfy domain-specific safety requirements, one essential safety requirement is that their virtual synchrony must always be maintained.

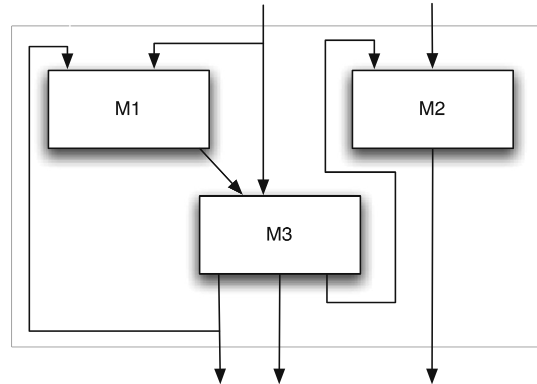
The design, verification, and implementation of these systems is quite *hard* and *error-prone* due to insidious *race conditions* that can occur because of asynchronous communication, message delays, and clock skews. In particular, their verification by testing is very difficult and full of uncertainties, since slight timing variations in system executions may either fail to reveal important bugs, or may lead to the ominous *no fault found* situation, where a known but unidentified bug cannot be caught after intensive testing, because the exact timing circumstances that caused the bug cannot be reproduced. To make things worse, model checking verification is often *unfeasible*, due to the typically huge state space explosion caused by the system's concurrency. Yet, the safety critical nature of these systems often requires thorough verification and a costly certification process. That is, these are complex systems for which a brute force approach is very costly and may still leave designers and safety inspectors with considerable uncertainty about the actual correctness of the system.

I explain below how the PALS formal pattern proposed in [45] and formalized and proved correct in [39,40] can drastically reduce the system complexity of a DRTS which should achieve virtual synchrony, making its design, verification and implementation much simpler than with standard approaches, and ensuring correctness by construction.

4.2. Context: Ensembles of state machines and underlying infrastructure

A key advantage of the PALS pattern is that its acceptable inputs are quite simple: they are pairs $(\mathcal{E}, \Gamma)$, where $\mathcal{E}$ is a rewrite theory specifying a collection of interacting synchronous state machines, called an *ensemble*, and Γ are infrastructure-dependent performance parameters further described below. I first sketch the ensemble idea. All the mathematical details can be found in [40].

Specifically, an *ensemble* $\mathcal{E}$ of *state machines* [39,40] is a collection of machines (i.e., automata) that interact synchronously through specified input and output connections. The figure below describes a typical ensemble with three interconnected state machines:



That is, each state machine can receive several inputs from either an external environment (the input wires entering the outer box enclosing the ensemble) or from other machines, which are routed according to the ensemble's *wiring diagram*. After receiving such inputs, all machines then perform their transitions *in lock step*, sending their outputs to other machines or to the external environment (outputs to the environment are depicted as wires leaving the outer box at the bottom). But such outputs *only become available as inputs in their destination components at the next cycle*.

This means that the ensemble $\mathcal{E}$ as a whole behaves exactly as a *single state machine*, called its *synchronous composition*, which is precisely the machine enclosed in the outer box, once we hide all the contents inside such a box. It also means that the notion of a machine ensemble is *recursive*: each state machine can potentially be decomposed as the synchronous composition of smaller state machines.

The external *environment* can be specified as either another state machine, or, more abstractly, by giving some constraints restricting the possible inputs that the environment may produce.

The assumptions about the *underlying infrastructure* necessary for the application of the PALS pattern are quite reasonable. They include: (i) an *asynchronous bounded delay* (ABD) network infrastructure, ensuring that there is a maximum bound on the communication delays between any two components; and (ii) a *clock synchronization algorithm* guaranteeing a maximum bound on clock skews. Based on such an infrastructure, the following *performance bounds* Γ can be derived on:

- Maximum *clock skew* ϵ with respect to a global clock;
- α_{min} and α_{max} times for each component reading its inputs, performing a transition, and producing outputs;
- μ_{min} and μ_{max} message *transmission delay* times.

The performance parameters Γ are used, together with the given ensemble $\mathcal{E}$, to form an acceptable input $(\mathcal{E}, \Gamma)$ for the PALS pattern.

The great simplicity and drastic reduction in system complexity afforded by the PALS methodology comes from reducing the design, verification and implementation of a complex, virtually synchronous DRTS to the much simpler task of designing and verifying the ensemble $\mathcal{E}$ which is its abstraction and computing the performance parameters Γ for the given infrastructure. In this way, as further explained in Section 4.3, the design of the DRTS itself is completely automated by the PALS transformation.

4.3. Solution: The PALS formal pattern

PALS stands for *Physically Asynchronous* yet *Logically Synchronous* systems. It is a formal pattern that *reduces* the design, verification and implementation of a DRTS that must achieve virtual synchrony to its much simpler *synchronous abstraction*, where the system operates in lock step and all complexities associated with asynchronous network communication, clock skews, and network delays are abstracted out.

PALS is a *theory transformation with performance parameters*

$$(\mathcal{E}, \Gamma) \mapsto \mathcal{A}(\mathcal{E}, \Gamma)$$

where, as described in Section 4.2, $\mathcal{E}$ is a *synchronous ensemble* of state machines, and Γ are *performance bounds* based on the underlying infrastructure. $\mathcal{A}(\mathcal{E}, \Gamma)$ is then the resulting asynchronous design of the distributed real-time system which, as further explained in Section 4.4, is a *correct by construction* implementation of $\mathcal{E}$ ensuring *virtual synchrony*, provided the performance requirements Γ are met by the underlying infrastructure.

The PALS asynchronous model. The *distributed* system $\mathcal{A}(\mathcal{E}, \Gamma)$, adds a "wrapper" meta-object around each state machine in $\mathcal{E}$. Besides containing the state of its state machine, each wrapper has: (i) an *input buffer* for messages arrived during the round; (ii) an *output buffer* to hold outgoing messages until they can be sent to the network; and (iii) *round* and *backoff timers* to start each cycle and send outputs at the right times.

The asynchronous system consisting of these wrapped state machines communicating by message passing can achieve logical synchrony with a period T determined by the performance constants Γ as follows: $T \geqslant \mu_{max} + 2 \cdot \epsilon + max(2 \cdot \epsilon - \mu_{min}, \alpha_{max})$, where this inequality is best possible [39,40].

The behavior of each node's wrapper is described by rewrite rules specified in Real-Time Maude [39] and can be summarized as follows:

1. At the beginning of each PALS period T (as perceived by the local clock): (i) reads messages from its input buffer; (ii) executes a local transition (change state and generate new messages); (iii) puts the new messages in its output buffer with a *back-off delay*.
2. Messages from the output buffer are sent to the network when the backoff timer has expired *and* the execution of the local transition has finished.
3. Received messages (for the next cycle) are put in the input buffer.

4.4. Guarantees: Main correctness result

The main correctness result for PALS is that the synchronous model given by the ensemble $\mathcal{E}$ with environment constraints c and its corresponding asynchronous implementation $\mathcal{A}(\mathcal{E}, \Gamma)$ are *bisimilar* [39,40]. Mathematically, this is expressed by means of Kripke structures $(\mathcal{E}_{c_e}, L)$ and $(\mathcal{A}(\mathcal{E}, \Gamma), L')$ respectively associated to the synchronous and asynchronous models (so that state predicates can be specified by the corresponding labeling functions L and L'). Also, in $\mathcal{A}(\mathcal{E}, \Gamma)$ a class of states called *stable* states is identified. These are states where each wrapper component has advanced enough in the given cycle so that the global distributed state can be abstracted out as a state of the synchronous system $\mathcal{E}$ by an abstraction function *sync*.

A key corollary of the bisimulation theorem is that $\mathcal{E}$ and $\mathcal{A}(\mathcal{E}, \Gamma)$ satisfy the *same temporal logic formulas*, when they are properly restricted to the stable states in $\mathcal{A}(\mathcal{E}, \Gamma)$. More precisely, the stable states of the asynchronous system $\mathcal{A}(\mathcal{E}, \Gamma)$ determine a Kripke structure $(Stable\mathcal{A}(\mathcal{E}, \Gamma), synch; L)$ such that the abstraction function

$$synch : \big(Stable\mathcal{A}(\mathcal{E}, \Gamma), synch; L\big) \longrightarrow (\mathcal{E}_{c_e}, L)$$

mapping stable distributed states to synchronous ensemble states defines a *bisimulation* between the two Kripke structures. As an immediate corollary (see [39,40]), for any formula $\varphi \in CTL^*(AP)$ on atomic propositions AP, and for any stable initial state $\{C_0; t_0\}$ of $(Stable\mathcal{A}(\mathcal{E}, \Gamma), synch; L)$, with C_0 its configuration of objects, and t_0 its global clock, we have the equivalence:

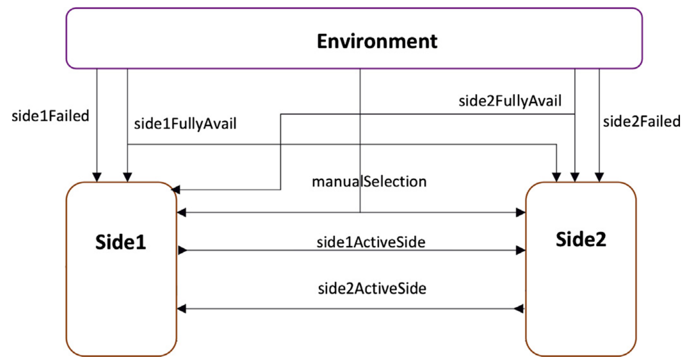
$$\big(Stable\mathcal{A}(\mathcal{E}, \Gamma), synch; L\big), \{C_0; t_0\} \models \varphi \quad \Leftrightarrow \quad (\mathcal{E}_{c_e}, L), synch\big(\{C_0; t_0\}\big) \models \varphi.$$

This is a very useful result, since it reduces the model checking verification of a property in $(Stable\mathcal{A}(\mathcal{E}, \Gamma), synch; L)$ to the verification of the same property in the much simpler and much smaller synchronous abstraction $(\mathcal{E}_{c_e}, L)$. As illustrated below with a case study, the state space reduction achieved by the synchronous abstraction can be so drastic that it allows verification of properties on the synchronous system that cannot be model checked on the asynchronous system due to state space explosion.

4.5. An avionics case study

The usefulness of the PALS bisimulation theorem can be illustrated by applying PALS to the following avionics case study presented in [40], which is based on an AADL model by Abdullah Al-Nayeem of a similar specification developed by Steve Miller and Darren Cofer. This case study, an *Active Standby System*, is typical of systems in integrated modular avionics, where there are, for fault tolerance reasons, multiple cabinets (with power supply, computer, I/O, etc.) distributed in an aircraft, so that the system has to be implemented as a DRTS.

The Active Standby System should decide which cabinet is *active*: (i) the *standby* side monitors the side's functionalities and the pilot's manual switch; and (ii) the *standby* side notifies the *active* side if change of active sides is needed. Furthermore, the system must be able to cope with *failures*; although it is assumed that the two sides will not fail simultaneously. The architecture of the *active standby system* for *two* sides can be summarized as follows:



Such an architecture can be naturally abstracted as a *synchronous design*, but in practice it must be realized as a *distributed* system of cabinets in the aircraft. It is therefore a design to which we can apply the PALS transformation to obtain a correct-by-construction distributed realization based on the performance parameters Γ supplied by the underlying ABD network, clock synchronization infrastructure, and so on.

The number of reachable states in the two models and the corresponding execution times can be compared by model checking an invariant. Specifically, the invariant that when a side is failed it will only transmit the value 0 is used.

In the *synchronous model*, the number of states reachable from the initial state is *185*, and model checking the invariant in Maude takes less than a second on a 1.86 GHz server with 8 GB RAM. In the *asynchronous model* the invariant was first model checked with *no messaging delay* in 33 minutes, exploring *3,047,832 reachable states*. If the environment is restricted to 12 instead of 16 possible different outputs (by not allowing side 1 to fail), then there are *1,041,376 reachable states*, and it takes about *190 seconds* to search the entire state space. If the environment is further restricted so that *no side can fail*, then there are *243,360 reachable states* and the invariant can be analyzed in *30 seconds*.

The asynchronous model was then analyzed for *maximal messaging delay 1*. That is, each message may take either zero or one time units to arrive. In this case, exhaustive state space exploration was *aborted* by the operating system after two hours. Restricting to 12 environment possibilities showed that *1,496,032 states* were reachable from the given initial state. The analysis took *420 seconds* of cpu time. With only eight different environment possibilities, the numbers were *349,856 reachable states* and *52 seconds*.

The following table summarizes these experiments:

Model	Max. msg. dly	8 env. possibilities		12 env. poss.		16 env. poss.	
		# states	ex. time	# states	ex. time	# states	ex. time
Synchr.	n/a	47	0.04 s	107	0.1 s	185	0.2 s
Asynchr.	0	243,360	30 s	1,041,376	190 s	3,047,832	2000 s
Asynchr.	1	349,856	52 s	1,496,032	420 s	aborted	

4.6. Advantages and shortcomings

The obvious advantage of PALS is the drastic reduction that it achieves in the complexity of a DRTS design, verification, and implementation. By allowing expressing a DRTS in terms of its much simpler, synchronous version, a design becomes much easier to understand, and its synchronization requirements are made explicit. At the verification level, particularly for model checking verification, the difference in many cases is one between being able to model check a synchronous design with much fewer states, or model checking verification being out of the question due to state space explosion. Regarding implementation, since PALS itself is a formal executable specification, automatic generation of synchronous code from the synchronous design is relatively straightforward, and the code generator is amenable to formal verification once a given target language with a precise formal semantics has been chosen.

However, not all DRTS design problems can be solved by PALS. To begin with, only control systems requiring virtually synchronous behavior and having components physically close enough to each other so as to make such virtual synchrony possible within required performance bounds are amenable to a PALS solution. Furthermore, PALS is based on the strong assumption that *all components operate at the same rate*. In practice this assumption is quite restrictive: the typical situation is that of *multirate* systems, where different components change at different rates but still need to be synchronized. Section 6.2 explains how PALS can be composed with two other simpler patterns to achieve a much more flexible pattern supporting multirate DRTS designs.

5. Examples 3–5: Reflective meta-object patterns for DoS protection

The examples in this section illustrate formal patterns based on distributed object reflection, using both onion-skin meta-objects and nested "Russian doll" meta-objects (see Section 2.2.3). A common theme running through these patterns is *network security*; specifically, protection against Denial of Service (DoS) attacks. The Server Replicator pattern of Section 5.3, a pattern to maintain server performance under heavy loads, is generally useful for QoS purposes but, as explained in Section 6.1, plays a crucial role in DoS protection when composed with the ASV pattern.

5.1. The cookies meta-objects

5.1.1. Problem: Modular DoS cookie protection

Usually, DoS protection mechanisms are not described and implemented in a modular way. Instead, an underlying protocol is changed in an ad-hoc manner to make it DoS-resilient. For example, the TCP protocol can be defended against SYN attacks using SYN cookies [13]. The main question asked and answered in [17] is: can cookie-based DoS protection be made modular and generic by formally specifying cookies as a pattern applicable to a wide class of systems?

5.1.2. Context: Client–server systems

The "context" part of the pattern is very simple: it can be applied to any input rewrite theory specifying a client–server system under minimum assumptions, namely, that the communication pattern is that of requests sent from clients to servers, which are answered back by corresponding replies from severs to clients. The specific nature of the requests and replies and the application domain of the given client–server system are totally irrelevant.

5.1.3. Solution: The cookie meta-objects

The solution is a formal pattern applicable to any client–server system under the minimal assumptions described above. The cookies pattern encapsulates the clients and servers in the given client–server system within two generic onion-skin meta-objects: one used to wrap each client, and another used to wrap each server. These meta-objects mediate the client–server communication by exchanging cookies and checking cookie credentials as a prerequisite for getting an answer from the server. The precise executable semantics of these meta-objects is given by the following rewrite rules, where _;_ denotes the multiset union operator used to form configurations containing clients, servers, and messages.

Client Wrapper Rules

ConnectReq: $[C, CP_c, \text{null}, (\text{to } S, m \text{ from } C); X]_c \to [C, CP_c, (\text{to } S, m \text{ from } C), X]_c; (\text{to } S, \text{connect from } C)$ if $\text{SIn}(S, CP_c) = \text{false}$

SetCookie: $(\text{to } C, k \text{ from } S); [C, CP_c, (\text{to } S, m \text{ from } C), X]_c \to (\text{to } S, (k, m) \text{ from } C); [C, \{(S, k)\} \cup CP_c, \text{null}, X]_c$ if $\text{SIn}(S, CP_c) = \text{false}$

MsgToServer: $[C, \{(S, k)\} \cup CP_c, \text{null}, (\text{to } S, m \text{ from } C); X]_c \to [C, \{(S, k)\} \cup CP_c, \text{null}, X]_c; (\text{to } S, (k, m) \text{ from } C)$

AcceptReply: $(\text{to } C, m_1 \text{ from } S); [C, CP_c, msg, X]_c \to [C, CP_c, msg, (\text{to } C, m_1 \text{ from } S); X]_c$ if not $m_1 : \text{Cookie}$

Service Wrapper Rules

CookieGeneration: $(\text{to } S, \text{connect from } C); [S, CP_s, l, Y]_s \to (\text{to } C, \text{rand}(l) \text{ from } S)\ [S, \{(C, \text{rand}(l))\} \cup CP_s, \text{next}(l), Y]_s;$ if $\text{CIn}(C, CP_s) = \text{false}$

ResendCookie: $(\text{to } S, \text{connect from } C); [S, \{(C, k)\} \cup CP_s, l, Y]_s \to [S, \{(C, k)\} \cup CP_s, l, Y]_s; (\text{to } C, k \text{ from } S)$

ForwardRequest: $(\text{to } S, (k, m) \text{ from } C); [S, \{(C, k)\} \cup CP_s, l, Y]_s \to [S, \{(C, k)\} \cup CP_s, l, (\text{to } S, m \text{ from } C); Y]_s$

DropRequest1: $(\text{to } S, (k, m) \text{ from } C); [S, CP_s, l, Y]_s \to [S, CP_s, l, Y]_s$ if $\text{CIn}(C, CP_s) = \text{false}$

DropRequest2: $(\text{to } S, (k, m) \text{ from } C); [S, \{(C, k)\} \cup CP_s, l, Y]_s \to [S, \{(C, k)\} \cup CP_s, l, Y]_s$ if $(k \neq k')$

ForwardReply: $[S, CP_s, l, (\text{to } C, m \text{ from } S); Y]_s \to [S, CP_s, l, Y]_s; (\text{to } C, m \text{ from } S)$

Let me explain what the above rewrite rules do. The meta-object for clients is used to wrap each client configuration X containing a client named C and possibly incoming or outgoing messages as follows[8]:

$$[C, CP_c, msg, X]_c$$

where C is the client's name, which is also used as the name of its meta-object wrapper, CP_c is a set of pairs of the form (S, k), where S is the name of some server, and k is a cookie that was provided by that server, *msg* is either a message, or the *null* constant, and X is the inner configuration containing the client named C and possibly some messages. The predicate $SIn(S, CP_c)$ checks whether a pair of the form (S, k) already exists in CP_c for server S.

[8] Note the abbreviated notation for objects. Following the convention in Section 2.2.3, we would use explicit names for the different attributes, say, $[C, cookies : CP_c, message : msg, conf : X]_c$. In the abbreviated notation the attribute names are left implicit.

Similarly, a meta-object for servers is used to wrap each server configuration Y containing a server named S and possibly incoming or outgoing messages as follows:

$$[S, CP_S, l, Y]_S$$

where S is the server's name, which is also used as the name of its meta-object wrapper, CP_S is a set of pairs of the form (C, k), where C is the name of some client, and k is a cookie already generated by the server wrapper to communicate with C's wrapper, l is a seed used to generate fresh cookies, and Y is the inner configuration containing the server named S and possibly some messages.

The conditional rule *ConnectReq* detects that a request message m from client C to server S exists in the wrapper's inner configuration, but that no cookie from S exists in CP_C, and no message is stored in the wrapper's third component. Instead of sending the message to the server, a request for connection is sent to the server's meta-object wrapper, while the message itself is stored in the wrapper's third component.

The reaction of S's server wrapper when receiving a connection request from C's client wrapper is described in the conditional rule *CookieGeneration*, where, after checking that no cookie already exists for C in CP_S, the random cookie $rand(l)$ is generated from the current seed l and is both stored with C in CP_S and sent to C's wrapper, while the seed l is updated to $next(l)$. Similarly, the rule *ResendCookie* takes care of the case when a connection request from C is received after a cookie for C had already been generated and just resends that cookie.

Upon receiving the cookie from S, the meta-object for client C both stores such a cookie, together with the name S, in its CP_C component and forwards to S's meta-object the request that was holding in its third component (together with the cookie) as specified in rule *SetCookie*. After a cookie for C has been stored in both C's wrapper and in the wrapper for the server S contacted by C, communication between the wrapped client and server can proceed normally. The rule *MsgtoServer* sends a request m from C, accompanied by the corresponding cookie k. The rule *ForwardRequest* passes on such a request to the server stored in the meta-object's inner configuration, provided the cookie received is the correct one. If it is not, either because no such cookie exists or because it is false, the request is dropped by the server's meta-object wrapper as specified by rules *DropRequest1* and *DropRequest2*. Obviously, these two rules are the heart of the DoS protection provided by the cookie mechanism.

Finally, the reply to C from server S appearing in S's inner configuration is sent by S's wrapper to C's wrapper according to rule *ForwardReply*; and it is passed down to C's inner configuration by C's wrapper according to the conditional rule *AcceptReply*, which performs a type check to ensure that the type of the reply is a normal message and *not* a cookie.

Provided the interactions of the original system follow the standard client–server communication protocol, the cookie meta-objects specified by the above rewrite rules will automatically make such a system DoS resilient without changing in any way its underlying functionality: the original clients and servers remain blissfully ignorant of the fact that they have been wrapped and know nothing about cookies.

5.1.4. Guarantees: Preservation of safety properties

The guarantees provided by the cookie meta-objects have two aspects. A first, obvious aspect is the actual DoS resiliency added, which is a quantitative availability property whose performance can be formally analyzed, for example by statistical model checking methods such as the ones discussed later in this section. A second, very important aspect of the pattern's guarantees answers the question: if the client-server system satisfied some safety properties *before* the cookie patterns were added, are such properties *preserved* after the cookie meta-objects are installed? The answer is *yes*, even in the presence of a Dolev–Yao intruder.

The formal guarantees for this second aspect take the form of two main *preservation theorems* for the cookie meta-objects as follows (see [17] for a more detailed description of these preservation theorems):

1. Any client–server system meeting the minimal assumptions of the cookie meta-object pattern and deployed in a faulty network environment satisfies the *same* safety temporal logic properties (without the $\bigcirc$ operator) as the system obtained by applying the cookie meta-objects to it.
2. Any client–server system meeting the minimal assumptions of the cookie meta-object pattern and deployed in an insecure network environment with a Dolev–Yao attacker satisfies the *same* safety temporal logic properties (without the $\bigcirc$ operator) as the system obtained by applying the cookie meta-objects to it.

In particular, if the original client–server system used a cryptographic protocol that had been verified secure against certain types of Dolev–Yao attacks, then all such safety properties remain valid when the system is hardened against DoS attacks by adding the cookie meta-objects.

5.1.5. Advantages and shortcomings

The most obvious advantage of this pattern is that it makes the cookie DoS protection mechanism fully modular and therefore available to any client–server system under minimal assumptions. Furthermore, the modularity achieved is not just the complete separation of concerns between the normal client–server communication and the cookie mechanism, but also the *modularity of safety properties*. That is, the safety properties of the original client–server system are fully preserved

by the pattern. The limitations of the pattern are those of the cookie mechanism itself, which provides a measure of DoS protection, but not the strongest DoS protection possible. For example, it implicitly assumes that the addresses of malicious attackers are fake, so that only honest clients will be able to receive and store cookies from the server they have contacted.

5.2. The Adaptive Selective Verification (ASV) meta-objects

5.2.1. Problem: Adaptive defense against DoS attacks

Security does not come for free: security defenses have themselves a cost. It can be expensive to use a DoS defense mechanism when no DoS attack exists. Cost can itself be part of the defense idea, in the sense that participants incur a cost for a service in terms of, for example, bandwith, computation time, or actual money, a cost that dishonest participants may not want or may not be able to pay. But such a cost still has to be payed by honest participants. When should such a price be payed? And how much should be payed? An insidious problem is that a DoS attack may be mounted in a stealthy, gradual fashion, so that a system's availability is ultimately crippled, but the worsening of conditions is so gradual that it may not be detected, or may be detected quite late. Is it possible to design DoS defenses that can *adapt* to the level of the DoS attack so that: (i) their cost is minimal when no attack exists; (ii) gradual increases on the severity of a DoS attack are detected and reacted to accordingly; and (iii) the cost of the defense is commensurate with the severity of the attack? Also, can such defenses be made *modular*, so that they can be easily applied to a wide range of systems?

5.2.2. Context: Client–server systems

As for cookies, the context part of the ASV pattern is very simple: it can be applied to any input rewrite theory specifying a client–server system under the minimal assumption that the communication pattern is that of requests sent from clients to servers, which are answered back by corresponding replies from severs to clients.

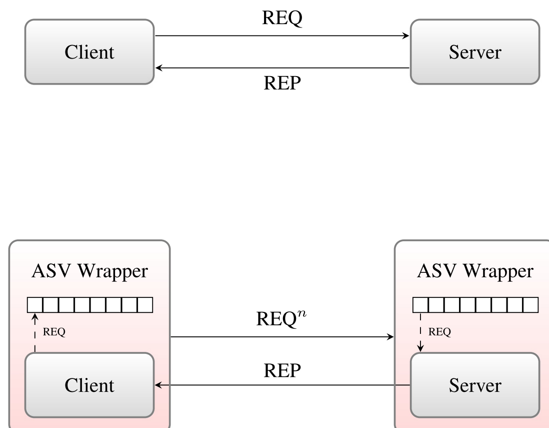
5.2.3. Solution: The Adaptive Selective Verification (ASV) pattern

The Adaptive Selective Verification (ASV) protocol [34] is a cost-based, DoS-resistant protocol which uses bandwidth as its currency. Its attacker model is a shared channel model [33]. It can be intuitively described by means of the following analogy. A DoS attack is analogous to somebody (a client) asking a question and trying to get a response from somebody else (a server) in a horribly noisy environment: communication can break down because the extreme noise drowns the question, which may not even be understood by the person to whom it is addressed. ASV is analogous to endowing the client with a megaphone whose intensity can be gradually increased as the level of noise increases; and endowing the server with noise cancellation technology that can tune out even extreme background noise, so that requests from honest clients can still be heard.

In ASV this is achieved by: (i) *adaptation*: the client resends an exponentially growing number of copies of its original request (up to a certain bound) to the server as it has to wait longer and longer to get a reply; and (ii) *selection*: the server probabilistically drops more and more requests as its becomes more and more congested because of an attack.

ASV has been formally specified as a probabilistic rewrite theory in [9], and has been explicitly described in [6] and [27] as a *formal pattern*, where client and server onion-skin meta-objects can be added to the objects of an underlying client–server system under minimum requirements *with no change whatsoever to the underlying system*.

The client–server system before and after applying the ASV pattern can be depicted as follows.



For example, the *selection* feature of the server meta-object allowing the server to probabilistically drop more requests as congestion increases is specified by the probabilistic rewrite rule *PR* below. Before explaining that rule, I explain what a probabilistic rewrite rule is in general. Usual rewrite rules $t \to t'$ are *deterministic*, in the sense that the set of variables appearing in the term t' is a subset of the set of variables in the term t. This means that when t is instantiated by a

matching substitution θ in a subterm u of a term $C[u]$, so that $u = \theta(t)$ and the term $C[u]$ can be rewritten by the rule to the term $C[\theta(t')]$, the resulting replacement subterm $\theta(t')$ is *fully determined* by the matching substitution θ. Instead, if there are extra variables in t' that do not appear in t, so that, say, the rule is of the form $t(\vec{x}) \to t'(\vec{x}, \vec{y})$, with $\vec{y}$ the extra variables in t', then the rule is *non-deterministic*. This is because the matching substitution θ for the variables $\vec{x}$ in t does *not* determine the result of applying the rule to a term $C[u]$ with $u = \theta(t)$: we still need to also instantiate the extra variables $\vec{y}$ in t' by another substitution ρ to get a resulting term $C[(\theta \cup \rho)(t')]$. In general there may be an infinite number of choices for ρ, so that the non-determinism may be infinitely branching.

A *probabilistic rewrite rule* [4] is a non-deterministic rewrite rule $t(\vec{x}) \to t'(\vec{x}, \vec{y})$ with the extra information that a probability distribution $\pi(\theta)$ is used to choose the substitution ρ for the extra variables $\vec{y}$, where the probability distribution $\pi(\theta)$ need not be fixed, but may in general depend parametrically on the matching substitution θ for the variables $\vec{x}$ in the left-hand side t. Such a probabilistic rewrite rule is then specified with the notation:

$$t(\vec{x}) \to t'(\vec{x}, \vec{y}) \textit{ with probability } \vec{y} := \pi(\theta).$$

In particular, if $\pi(\theta)$ is an independent distribution, the vector assignment $\vec{y} := \pi(\theta)$ can be decomposed as a conjunction $y_1 := \pi_1(\theta) \wedge \cdots \wedge y_n := \pi_n(\theta)$.

Let me now explain the probabilistic rewrite rule *PR* below.

$$\begin{aligned}&PR : \big(s \leftarrow (c, r)\big)[s, cnt : n, buf : L, conf : X] \to \textit{if } b \textit{ then}\\ &\big[s, cnt : n+1, buf : L[k] := (c, r), conf : X\big] \textit{ else } [s, cnt : n+1, buf : L, conf : X] \textit{ fi}\\ &\textit{if } |L| \geqslant MaxLoad\\ &\textit{with probability } b := Bernoulli\big(f(n+1)\big) \wedge k := Unif\big(|L|\big).\end{aligned}$$

The rule *PR* is conditional, so that it is only applied if the size $|L|$ of the buffer L exceeds a maximum load *MaxLoad*. It specifies the probabilistic behavior of the server wrapper $[s, cnt : n, buf : L, conf : X]$ containing the actual server named s in its inner configuration X, and having a buffer L to store pending requests and a request counter n, when it receives a request message $(s \leftarrow (c, r))$ from client c with contents r and the maximum load has been exceeded.

Note that an if-then-else operator is used in the right-hand side term $t'(\vec{x}, \vec{y})$; if-then-else is an equationally-defined function like any other one and has the standard semantics. Note also that extra variables $\vec{y}$ in the right-hand side are the Boolean variable b, and the natural-number variable k. The value of b is chosen by tossing a coin with bias determined by a function $f(n+1)$ whose details are unimportant. That is, the choice is made according to the Bernoulli distribution $Bernoulli(f(n+1))$. Instead, the value of k is chosen in the set of natural numbers up to $|L|$ according to the uniform distribution. If b comes out *true*, then the counter is increased, and the request in the k-th position of the buffer is discarded to make room for the new request from c. Otherwise, the new request from c is ignored, but the counter is increased.

5.2.4. Guarantees: Adaptive DoS protection

What formal guarantees can be given for the ASV pattern? Some first answers to this question were given in [34] by analytic hand proofs under some simplifying assumptions. The answers show that ASV is somehow *optimal*, in the sense of leading to a DoS defense by clients and servers that *very closely approximate the behavior of an omniscient DoS defense protocol*. An "omniscient" DoS defense protocol is one in which clients and servers are assumed to have *complete knowledge* of two key rates: (i) the *client request factor* ρ, measuring the proportion of the server's computation resources consumed by honest clients per time window; and (ii) the *attack factor* α, measuring the proportion between the total amount of computation required to process all requests sent by the attacker on a given time window and the total amount of computation that a server is capable of in such a window. Obviously, if $\alpha > 1$, the server is already overwhelmed; and if $\alpha \gg 1$, an unprotected server can be so overwhelmed that a DoS attack will fully succeed in shutting it down. In the idealized omniscient DoS protocol described in [34], clients and servers adapt to the attack having full knowledge of the values α and ρ at any time. Assuming that α and ρ cannot exceed respective upper bounds, using the omniscient protocol any client can connect with its server with very high probability (Theorem 1 in [34]). Of course, such omniscience is impossible to achieve in practice, due to the distributed nature of the problem. What a client observes in ASV is the latency in getting an answer back for its own requests; and what the server observes is essentially $\alpha + \rho$. However in ASV: (i) each client succeeds in establishing a connection with essentially the same confidence guarantee as in the omniscient case (see Theorem 2 in [34]); and (ii) the bandwith consumption needed by clients adapting to the attack is essentially of the same order as that required in the omniscient protocol (see Theorems 3–4 in [34]).

By formally specifying both ASV and the omniscient protocol as probabilistic rewrite theories and performing statistical model checking verification on both theories, the above results have been extended in [9] as follows: (i) the hand proofs of Theorems 1, 2, and 3 have been confirmed by model checking verification; (ii) the same results have been shown to hold beyond the bounds assumed by the hand proofs by model checking verification; and (iii) comparisons with non-adaptive DoS defense protocols that in [34] were based only on the evidence provided by simulations have been verified to hold with high statistical confidence by model checking verification.

The general idea is that analytical hand proofs become increasingly difficult as one considers more general and complex operating conditions. Under such, more complex conditions, simulations can give partial evidence about the behavior that a system will exhibit; but such an evidence can be weak, because a somewhat different choice of parameters might yield a simulation showing a quite different behavior. Statistical model checking verification occupies a useful middle ground between hand proofs and simulations: weaker than hand proofs but of much more general applicability; and allowing considerably stronger conclusions than those one can glean from simulations results alone.

5.2.5. Advantages and shortcomings

Two main advantages are ASVs optimal, omniscient-like adaptability, and the genericity of ASV when specified as a pattern applicable to any client–server system under minimal assumptions. The main limitation is that, although ASV can optimally adapt to increasingly harsh DoS attacks and maintain a client–server system still available to clients with high probability, as the attack worsens, a significant service degradation in terms of average latency is unavoidable, because more and more copies of honest client requests get discarded by rule *PR* before each honest request gets answered. However, we shall see in Section 6.1 that, when the advantages of ASV are combined with those of the Server Replicator (SR) pattern, it is possible to maintain a fixed quality of service even when under increasingly harsh DoS attacks.

5.3. The Server Replicator (SR) meta-object

5.3.1. Problem: Provisioning more servers to handle high loads

Web-based systems can dynamically adapt to high-demand situations by dynamically allocating extra resources. For example, a server can adapt to increases in demand or to DoS attacks by leveraging the scalability of the Cloud to provision more copies of itself. Provided the needed resources are not exhausted, this can allow a service to continue its operation without any substantial loss in the quality of service. In extremely high-demand situations, however, such extra resources may be exhausted, or may become prohibitively expensive. Therefore, a tradeoff exists between the quality of service (QoS) offered and the amount and cost of resources dynamically provisioned.

5.3.2. Context: Simple client–server systems

The context part of the SR pattern is even simpler than in previous patterns; but such simplicity makes the pattern somewhat *restrictive* in its applicability. It can be applied to any input rewrite theory specifying what I call a *simple client–server system*. This is a client–server system such that: (i) its communication pattern is that of requests sent from clients to servers, which are answered back by corresponding replies from servers to clients; and (ii) no other intermediate messages are involved in the client–server communication: just a client request followed by a reply from the server. For example, a client–server system enhanced with the cookie wrappers of Section 5.1 is *not* a simple client–server system, since the request and the reply messages are themselves mediated by the messages from the corresponding wrappers to share a cookie. The SR pattern cannot handle a system like this, since it is essential to SR that the particular server clone handling a client request should be immaterial; but if cookie information has to be exchanged, this assumption is violated.

5.3.3. Solution: The Server Replicator (SR) pattern

The Server Replicator meta-object (SR) is a simple *Russian dolls* meta-object pattern developed in [26,47,27]. The SR pattern can be applied to the server side of any simple client–server system as described above. SR has an *overloading factor* parameter, beyond which a new server is provisioned. This parameter can be fine tuned to reach the desired tradeoff between QoS and cost of resource allocation. An overloading factor of 1 means that servers are already overwhelmed.

In more detail, the SR meta-object: (i) wraps around several instances of the same server; (ii) distributes incoming requests among the wrapped servers in a random, uniform way; (iii) keeps information about the load of the servers based on the amount of requests received and periodically estimates such load; and (iv) spawns a new server each time the estimated load exceeds the chosen overloading factor.

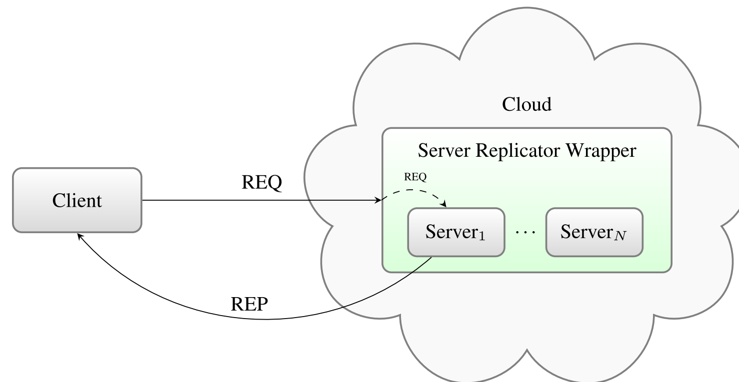
For example, the probabilistic rewrite rule governing how the SR meta-object forwards an incoming message to a randomly chosen replicated server with name i in its current server pool C is specified by the following *SC* probabilistic rewrite rule:

$$SC : \big(s \leftarrow (m, c)\big)[s, server.list : SL, conf : C] \rightarrow$$

$$\big[s, server.list : SL, conf : \big(i \leftarrow (m, c)\big)C\big] \; \textit{with probability } i := Random(SL)$$

where the name i of the server in the server pool C is chosen randomly according to a uniform distribution among the names in the server list SL.

The distributed architecture of SR is depicted below.



Provided that the overloading factor is below 1, so that the servers are never overwhelmed, the SR pattern can ensure a predictable and stable QoS from the replicated servers. However, as already mentioned, under severe high-demand situations such as those in a DoS attack, provisioning new servers so that they are never overwhelmed may require a very large number of servers and can be too expensive.

Could it be possible to find a way to raise the overloading factor way beyond 1 (so that we can make do with much fewer resources), yet in such a way that QoS is maintained? This question is explored in Section 6.1, where an answer based on combining SR with ASV is given.

5.3.4. Guarantees

Provided that: (i) the overloading factor is kept below 1; (ii) the time required for processing server requests is relatively uniform, so that message count is a reliable way of estimating server load; and (iii) the SR meta-object periodically checks the load with sufficient frequency, a stable quality of service, what [27] calls *stable availability*, can be maintained. Since, the statistical model checking verification for several QoS indicators has been verified for SR not just for overloading factors below 1, but for factors higher than 1 when combined with the ASV pattern, a detailed discussion of such verification is postponed until the more general setting of the ASV + SR pattern is presented in Section 6.1.

5.3.5. Advantages and shortcomings

The key advantage of SR is of course that it allows dynamic adaptation to increasingly heavier loads while maintaining a stable quality of service. The limitations are twofold. First, a stable quality of service can only be ensured if: (i) the overloading factor is below 1; and (ii) the total load does not exceed the number of hardware resources available to provision new servers, which can easily be exhausted if a massive DoS attack against the servers is launched. Second, only simple client–server systems where the only pattern of communication is a client request followed by a server reply are acceptable inputs for the SR pattern.

6. Examples 6–7: Composed patterns

The notion of composed pattern was already discussed in Section 2.3. As pointed out there, not all possible pattern compositions are generally useful. Therefore, attention should focus on compositions that solve important problems and are widely applicable. Lack of such usefulness may result from several reasons. First, some pattern compositions *do not preserve* the guarantees offered by some of the patterns used in their composition and therefore are *bad* compositions. Second, some arbitrary compositions may not be generally useful, because the combination of functionalities that they provide is too *ad-hoc* to meet a frequently occurring need. This does not make some such compositions useless: some may still be quite useful for a specific, concrete design having a particular combination of needs, and may even happen to preserve properties ensured by the patterns used in the composition. The point is that in such cases the given composition should be part of the *architecture* of a particular system, instead of being a highly reusable pattern by itself.

Of particular interest are compositions that: (i) solve a common problem and are applicable to a wide class of input systems; (ii) preserve the properties of the patterns used in the composition; and (iii) may even have *emerging properties* that cannot be achieved by any of the simpler patterns out of which they are built. If at least conditions (i)–(ii) are met, we

can rightfully call such compositions *composed patterns*. The two composed patterns presented in this section, namely, the ASV + SR pattern and the Multirate PALS pattern, do actually satisfy conditions (i)–(iii) and are therefore useful examples that illustrate the general idea.

6.1. The ASV + SR pattern: Stable availability under DoS attacks

Note that neither ASV nor SR are sufficient in practice to withstand a severe DoS attack while maintaining desired QoS levels. On the one hand, the problem with SR is that, unless the overloading factor remains below 1 to ensure that the servers are never overwhelmed, messages from honest clients may either get dropped, or may only be answered after long delays. But during a severe DoS attack, over-provisioning to always keep servers from being overwhelmed may easily become either too expensive or just unrealistic. On the other hand, the problem with ASV is that, although it ensures that with high probability all honest client messages will be answered, the average *latency* experienced by clients may easily increase beyond acceptable limits during a severe DoS attack. Before explaining how the SR and ASV formal patterns are combined in [26,47,27] to maintain acceptable QoS under DoS attacks, I explain the problem that ASV + SR solves, namely, that of achieving *stable availability* under very adverse conditions.

6.1.1. Problem: Achieving stable availability under DoS attacks

Availability is a key security property by which a system remains available to its users under some conditions. This property can be compromised by a DoS attack, which may render a system unavailable in practice. Since availability properties are quantitative properties, some DoS defense mechanisms may provide better availability properties than others. In fact, even when protected against DoS, performance degradation will typically be experienced in some aspects of system behavior such as, for example, the average Time To Service (TTS) experienced by clients, the success ratio with which clients manage to communicate with their server, or the average bandwidth (or some other cost measure) that a client needs to spend to successfully communicate with its server. Obviously, an ideal DoS protection scheme is one that renders the system to a large extent impervious to the DoS attack, no matter how bad the attack can get (up to a maximum bound assumed in the shared channel model). That is, up to some acceptable and constant performance degradation, the system behaves in a "business as usual" manner: as if no attack had taken place, even when in fact the attack worsens over time. This property is called *stable availability* in [27].

More precisely, stable availability is formulated as a requirement parameterized by explicitly specified and quantifiable *availability properties* such as, for example, TTS, success ratio, average bandwidth, and so on. The system is then said to be stably available with respect to the specified quantities if and only if, with very high probability, each such quantity q remains very close (up to fixed bounds ϵ) to a threshold θ, which does not change over time, regardless of how bad the DoS attack can get within the bounds allowed by the shared channel assumption.

6.1.2. Context: Simple client–server systems

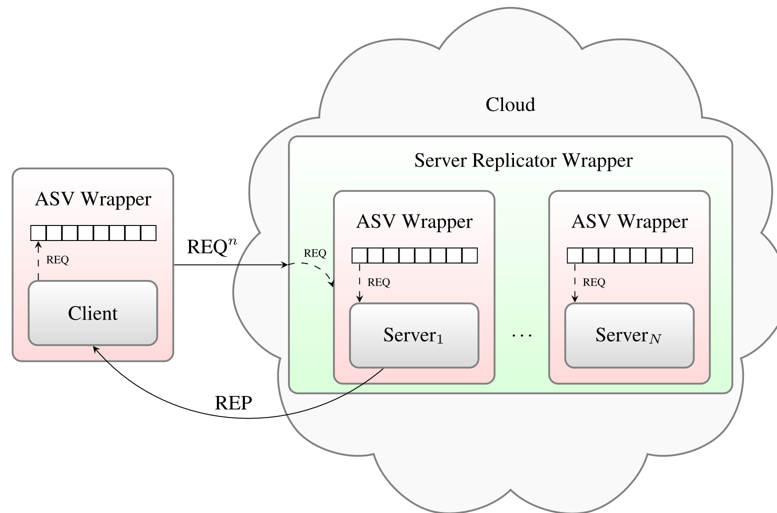
The context part of the ASV + SR pattern is the same as that of the SR pattern described in Section 5.3.2, namely, a *simple client–server system* such that its communication pattern is that of requests sent from clients to servers, which are answered back by corresponding replies from servers to clients with no other intermediate messages involved in the client–server communication. The key point is that the ASV meta-objects do not generate any meta-object-specific messages: a client ASV wrapper just keeps resending an increasing number of copies of a request message until it is answered; and a server ASV wrapper receives client requests (possibly dropping some) and, when they are answered by the server in its inner configuration, forwards their replies as usual. This means that a simple client–server system *remains simple* after being composed with the ASV pattern. As explained in Section 5.3.2, this is essential to be able to correctly compose ASV and SR.

6.1.3. Solution: The combined ASV + SR pattern

The idea of ASV + SR is very simple: to compose ASV and SR into a pattern that combines the advantages of both without their respective shortcomings, so that stable availability can be achieved. SR by itself becomes fragile as soon as servers reach their overwhelming level. But combined with ASV, the overload factor may safely be raised way above 1. This means that provisioning of resources under the extreme conditions of a DoS attack now becomes feasible, since much fewer servers are needed to achieve stable availability. This offers to a system designer a useful and easily tunable tradeoff between a somewhat higher QoS by provisioning more resources (but much fewer than those needed with SR alone), or a somewhat lower QoS by provisioning extra resources more greedily.

As explained in [47,27] and summarized below, ASV + SR is a *Russian dolls distributed formal pattern* applicable to any client–server system under minimum requirements and easily *tunable* by the overload factor of SR. ASV + SR is obtained by *composing* a client–server system with the ASV and SR patterns as follows: (i) clients and servers adapt to the DoS attacks

by being composed with *ASV onion-skin meta-objects*; and (ii) servers in the cloud adapt to an increased load with the *SR Russian dolls meta-object*. In this way, we obtain the *adaptive distributed architecture* depicted below.



6.1.4. Guarantees: Stable availability properties under DoS attacks

What formal guarantees can be associated to the ASV + SR pattern? In particular, which stable availability properties are ensured by ASV + SR? Also, how can such properties be verified?

This last question is nontrivial for two reasons: (i) both ASV and SR involve *probabilistic* algorithms and are in fact *probabilistic rewrite theories* [4], so standard model checking techniques are insufficient: one needs to use either probabilistic or statistical model checking techniques; and (ii) stable availability properties are intrinsically *quantitative*, so that neither a "true" or "false" answer, such as the answer given to a standard temporal logic formula, nor even a *probability* answer $p \in [0, 1]$ associated to a probabilistic temporal logic formula are meaningful. What we need is a *quantitative* answer in the form of a real number value (for example an average latency), or perhaps an *interval* $[a, b]$, within which such a quantity lies.

All this suggests expressing the desired stable availability properties in a *quantitative* temporal logic such as QuaTEX [4], where formulas evaluate not to "true" or "false," nor even to a probability $p \in [0, 1]$, but to an interval of real number values. Indeed, all the desired stable availability properties for ASV + SR can be naturally expressed by QuaTEX formulas and can be efficiently model checked in parallel using the PVeStA tool [8].

In [47,27], several stable availability properties of ASV + SR have been specified in QuaTEX and have been verified in PVeStA by statistical model checking with a 99% confidence interval. The following quantities can be maintained stable under an increasingly harsher DoS attack using the ASV + SR pattern:

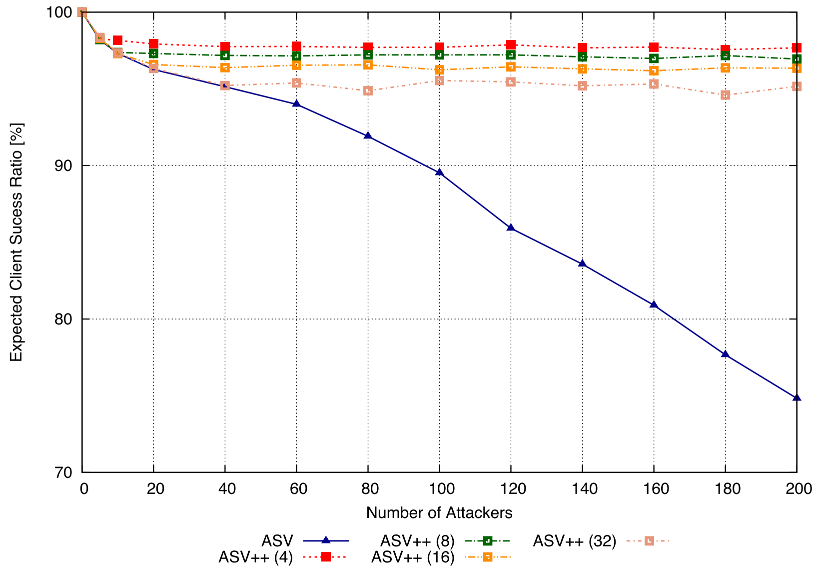
- The expected *client success ratio*, i.e., the percentage of clients that are able to establish a connection with the service within a given time bound.
- The expected *average TTS*, i.e., the average time it takes a client to connect to the service.
- The expected *client bandwidth*, i.e., the amount of bandwidth a client has to use to establish the connection.

Furthermore, the expected *number of servers* that need to be provisioned for the service in the cloud for a chosen *overload factor* was also analyzed by statistical model checking.

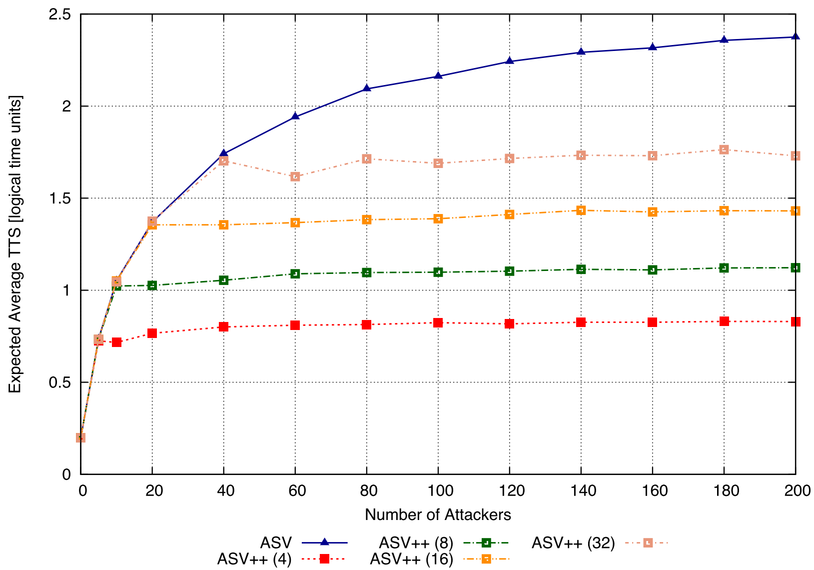
The results of the statistical model checking analysis reported in [47,27] are summarized in the following graphics, where in all cases the horizontal line describes the number of attackers (1.5 attackers are enough to completely overwhelm a server), and the vertical axis measures the desired quantity. The following choices of the overload factor were analyzed: $k = 4, 8, 16, 32$, where $k = 1$ is the level at which servers are overwhelmed. For example, ASV + SR(8) means that the overload factor has been set 8 times above the level when servers are overwhelmed. Also, for comparison purposes, the behavior of ASV without SR is plotted in some of the graphics.

ASV + SR can sustain the expected client success ratio above 90% at the cost of provisioning new servers. Note that this success ratio is measured *within a limited time window*: if one waits long enough, ASV ensures that requests will *eventually*

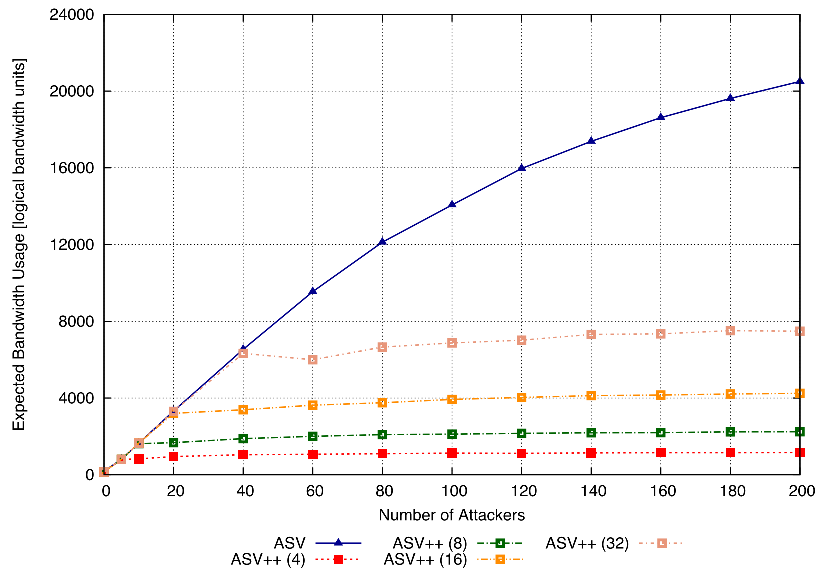
be answered with high probability. Note that the corresponding success ratio of ASV *within the given time window* is seen to drop as the DoS attack worsens.



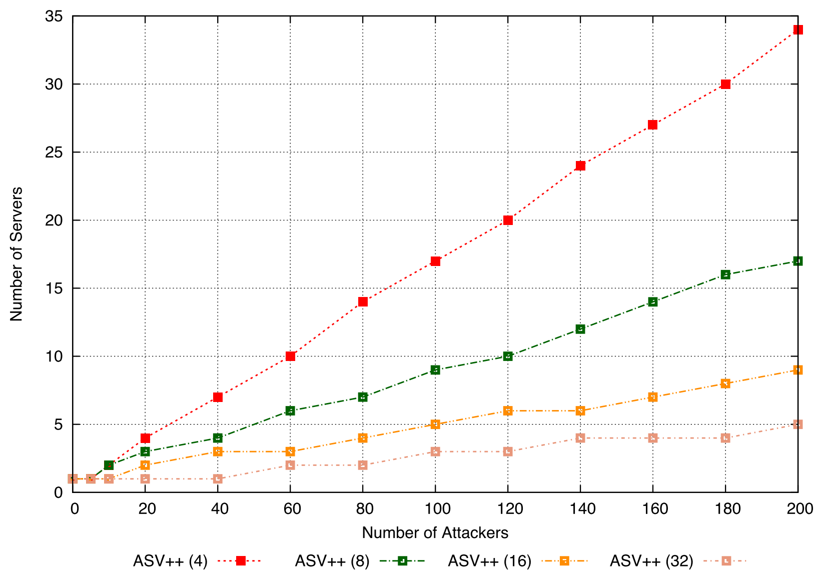
ASV + SR can sustain the expected average TTS at a low level. Instead, the TTS with ASV alone increases as the DoS attack worsens.



ASV + SR keeps the expected client bandwidth usage at a constant level, whereas with ASV alone client bandwith usage increases as the DoS attack worsens.



Higher overloading factors reduce the amount of provisioned servers significantly. This should make it possible to withstand a severe DoS attack with a realistic amount of resources and at reasonable cost.



6.1.5. Advantages and shortcomings

The main advantage of ASV + SR is that it combines the omniscient-like optimal adaptation of ASV as a DoS defense mechanism, ensuring that servers remain available with high probability, with the SR adaptation to higher server loads. By combining both patterns, stable availability under DoS attacks becomes possible. This was not achievable by ASV alone, and was only achievable at very high cost by SR with a very large amount of resources. The main limitation of the ASV + SR pattern is that the client–server systems to which it can be applied must be *simple*; that is, their communication sequences must be restricted to client requests followed by server replies.

6.2. The multirate PALS pattern

6.2.1. Problem: Achieving virtual synchrony for a multirate asynchronous DRTS

As pointed out in Section 4.6, the strongest limitation of the PALS pattern is that it assumes that *all components operate at the same rate*. In practice this assumption is too restrictive, because, for physical reasons, different sensors and actuators must

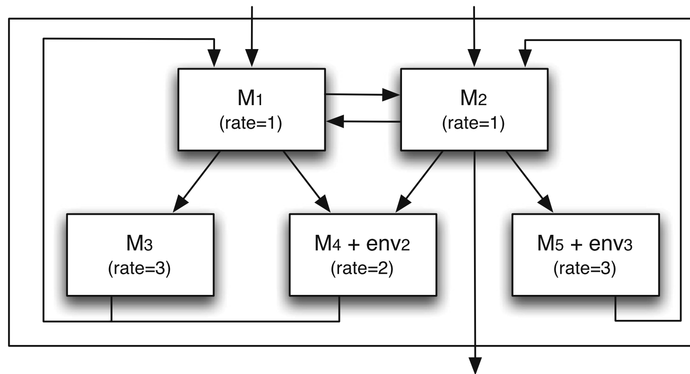
often operate at different rates. This means that in a distributed control system, subcomponents closer to the sensors and actuators are typically *faster* than intermediate components, which are themselves faster than a central controller. Of course, all these components still need to act in a *virtually synchronous* way. But now this synchronization must take account of the different *rates* (integer multiples of the slowest rate, namely, that of the central controller) at which different components operate.

If anything, the multirate nature of these systems makes them even more complex than a DRTS operating at a single rate. Can virtual synchrony be achieved for a multirate DRTS? Can the good features of PALS somehow be generalized to the multirate setting? Pattern composition seems to be the simplest and most satisfactory answer: we do not need to change PALS itself; we just need to compose it with appropriate patterns to handle the acceleration and deceleration involved in the multiple rates.

6.2.2. Context: Multirate ensembles and underlying infrastructure

As for PALS, we need first of all to specify the semantic conditions for the acceptable inputs to our desired Multirate PALS pattern. In this case, such inputs should be triples $(\mathcal{E}, T, \Gamma)$, where $\mathcal{E}$ is a *multirate* ensemble, T is the global period, and Γ are the performance bounds provided by the underlying infrastructure.

A *multirate ensemble* is a *synchronous interconnection* of state machines with specified rates, as the one depicted below (for a precise mathematical definition, see [11]):



where the two wires entering the outer box are those of the *slow environment* (for example, the airplane pilot interacting with the airplane's central control system), whereas some faster machines may have their own *faster environments* (env_2 and env_3 in the picture), corresponding for example to sensors and actuators. Since an environment can be viewed as another (typically non-deterministic) state machine, in the above picture we assume that M_4 has been composed with env_2 into a single machine, and likewise that M_5 has been composed with env_3, so that, conceptually, only the slow environment is now outside the outer box.

However, the above picture is still too high level and leaves a number of important questions unanswered. For example, in what sense can the outputs of, say, the faster machine M_3 become inputs of the slower machine M_1? Conversely, in what sense can the outputs of, say, the slower machine M_1 become inputs of the faster machine M_3? These obvious questions are not entirely trivial. They can however be answered by introducing two simple *auxiliary patterns* for (typed) state machines:

1. The *k-decelerator pattern* is a transformation

$$(M, k) \mapsto M^{\times k}$$

with M a typed machine and k a nonzero natural number; and

2. The *adaptor closure* is a transformation

$$(M, \alpha) \mapsto M_{\alpha}$$

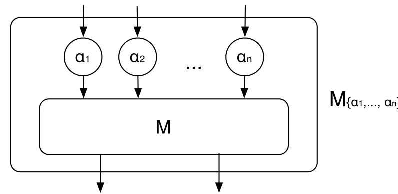
with M a typed machine and α a tuple of functions called an *input adaptor*.

The details of both transformations are spelled out in [11], but the key ideas are easy and natural. $M^{\times k}$ is the machine obtained by "running" M k times, so that M and $M^{\times k}$ have the same set of states, but the inputs to (resp. outputs from) $M^{\times k}$ are k-tuples of inputs to (resp. outputs from) M. Therefore, $M^{\times k}$ "skips all intermediate states" and transitions to the state reached after consuming k inputs. In the above multirate ensemble picture, for example, $M_3^{\times 3}$ now has the same rate as M_1, so that talk about feeding inputs and outputs to each other becomes meaningful.

There is, however, a remaining problem: inputs to $M_3^{\times 3}$ must now be k-tuples, and outputs form $M_3^{\times 3}$ are likewise k-tuples; but inputs and outputs to M_1 need not be k-tuples at all. This remaining problem is solved by the adaptor closure pattern.

The technical definition in [11] (and the above ensemble picture) assume that state machines may have several input and output wires, feeding different types of data; that is, they assume *typed* machines. Let us assume, for example, that the output wire of M_3 and the leftmost input wire to M_1 both carry a floating point number. Likewise, let us assume that the rightmost input wire of M_1, the leftmost output wire of M_1, and the rightmost input wire of M_3 all carry a Boolean value. How can we *adapt* these inputs and outputs? By introducing input adaptor functions for each machine.

For example, we can adapt the inputs of M_1 by using a pair of functions $\alpha = (\alpha_1, \alpha_2)$, with $\alpha_1 : (x_1, x_2, x_3) \mapsto (x_1 + x_2)/2$ (that is, we take the *average* of the first two inputs), and with α_2 the identity function on the Booleans. A function like α_1 may indeed be very useful. Why? Because in an actual distributed implementation, the last value x_3 from M_3 may, due to network delays and clock skew, *arrive too late* for it to be used by M_1. α_1 then gives us the flexibility of *ignoring* that last input from M_3 that M_1 cannot get on time anyway. Likewise, we can for example adapt the inputs of $M_3^{\times 3}$ by using a function $\beta : b \mapsto (b, \bot, \bot)$, where $\bot$ is a "no-op" input which we assume M_3 can also accept. That is, the only meaningful input is the first Boolean input b; the remaining two are no-ops. The adaptor closure transformation $(M, \alpha) \mapsto M_\alpha$ constructs a new machine where adaptors have first been added to the input wires, as depicted below (again, see [11] for the detailed definition):



In our two examples we get in this way input-adapted machines $M_{1\alpha}$ and $M_{3\beta}^{\times 3}$ that *now* can be synchronously composed in the usual sense.

In summary, and as further explained in [11], a multirate ensemble $\mathcal{E}$ contains information not only about the different machines, their rates, and their wiring, but also about the *input adaptors* needed to hook them together. Then, the *synchronous* composition of $\mathcal{E}$ is the single *slow* machine obtained by: (i) slowing down fast machines; and (ii) adding input adaptors to each machine to obtain a *single-rate ensemble*, just as in Section 4.2; then (iii) we obtain our desired synchronous composition of $\mathcal{E}$ as the slow machine that is the standard (in the sense of Section 4.2) synchronous composition of the single-rate ensemble obtained after steps (i) and (ii).

The performance bounds Γ are provided by the underlying bounded delay network and the clock synchronization algorithm, exactly as for PALS (see Section 4.2). The addition of a third argument T corresponding to the slow period is now important because of the presence of fast machines. In an asynchronous setting, if a distributed object encapsulating a k-fast machine is going to be able to deliver *all* its k messages on time as inputs to a slow machine, the inequality $T/k \geqslant 2\epsilon + \mu_{max} + max(2\epsilon - \mu_{min}, \alpha_{max,k})$ must be satisfied, where $\alpha_{max,k}$ is the maximum time needed by the k-fast machine to process inputs, make a transition and produce outputs. If k is large enough, satisfying this inequality with a desired period T may be unfeasible. Fortunately, however, as illustrated by the example of the adaptor function $\alpha_1 : (x_1, x_2, x_3) \mapsto (x_1 + x_2)/2$ above, for many applications it is *not* necessary that the above inequality is maintained. However, in the case of α_1, T still has to be such that the first *two* inputs from the distributed version of M_3 arrive on time. As further explained in [11], this means that, based on the bounds Γ, the chosen slowest period T and the ensemble's adaptor functions *co-determine each other*, and need to be checked against each other to ensure that the synchronous multirate design has a feasible asynchronous implementation.

6.2.3. Solution: The multirate PALS pattern

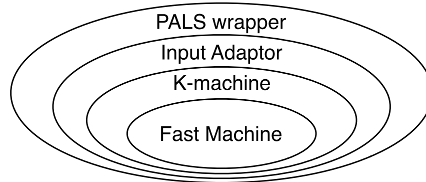
A decelerated machine $M^{\times k}$ is of course a very useful abstraction to understand a synchronous multirate design; but in the physical world of sensors and actuators nothing can be decelerated. This means that in an actual distributed implementation it is the k-fast machine M, and not its decelerated version, $M^{\times k}$, that has to be running. And yet, it is $M^{\times k}$ that should interact with the slow machines using suitable *adaptors*. How can this tension be reconciled? By developing *asynchronous versions* of the k-decelerator pattern and the adaptor closure pattern as *onion-skin wrappers* in Real-Time Maude. This has also the great advantage, both in terms of modularity and of conceptual clarity, of not requiring any change whatsoever to the original PALS wrappers, and is the key to the modular design of Multirate PALS as a pattern composition.

That is, Multirate PALS is a theory transformation of the form:

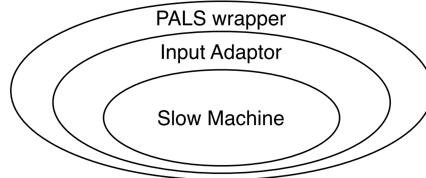
$$(\mathcal{E}, T, \Gamma) \mapsto \mathcal{MA}(\mathcal{E}, T, \Gamma)$$

where $(\mathcal{E}, T, \Gamma)$ satisfies the semantic requirements discussed in Section 6.2.2 (including the satisfaction of the mutual constraints that, given the performance bounds Γ, must be satisfied between the slow period T and the adaptors in $\mathcal{E}$).

The key idea is that each machine in the ensemble $\mathcal{E}$ is wrapped by several onion-skin wrappers. Specifically, a k-fast machine is wrapped by a 3-layer onion meta-object of the form:



whereas a slow machine is wrapped by a 2-layer onion meta-object of the form:



This means that all the multirate complexities are handled by the inner k-decelerator and input adaptor wrappers, so that the outer PALS wrapper can behave *exactly as if the system were a single-rate system*.

6.2.4. Guarantees: Main correctness result

Recall from Section 6.2.2 that, given a multirate ensemble $\mathcal{E}$, its synchronous composition is the single state machine obtained as the synchronous composition of the single-rate ensemble obtained by: (i) slowing down fast machines; and (ii) adding input adaptors to each machine. The key result about Multirate PALS presented in [11] is that, relative to a set of atomic propositions AP and a labeling function L, the Kripke structure associated to the synchronous composition of $\mathcal{E}$ and the Kripke structure determined by the *stable states* of $\mathcal{MA}(\mathcal{E}, T, \Gamma)$ are *bisimilar*, and therefore satisfy the *same CTL** formulas.

There is, however, a small technical proviso required for the above result to hold. It has to do with the fact that, thanks to the use of input adaptors, we may have distributed designs $\mathcal{MA}(\mathcal{E}, T, \Gamma)$ where not all k outputs of a k-machine reach a slow machine on time. For example, for the input adaptor $\alpha_1 : (x_1, x_2, x_3) \mapsto (x_1 + x_2)/2$ it is enough for inputs x_1 and x_2 to arrive to the slow machine M_1. We can model this by considering that the input entering M_1's input adaptor is the triple $(x_1, x_2, \perp)$. Since $\alpha_1(x_1, x_2, x_3) = \alpha_1(x_1, x_2, \perp)$, it does not matter at all that x_3 could not arrive on time. However, in the definition of the synchronous product of $\mathcal{E}$, outputs to be consumed in the next cycle *are part of the state*, so that the state containing the output (x_1, x_2, x_3) and that containing the output $(x_1, x_2, \perp)$ are *different* states. The technical proviso required for our desired bisimulation result to hold is that the labeling function L of the Kripke structure associated to the synchronous product of $\mathcal{E}$ *should not be able to distinguish* between these two states. Technically, this is formalized by defining an equivalence relation $\equiv_A$ of "adaptor equivalence" between states of the synchronous product, where two states are equivalent iff the outputs contained in the two states are equal after adaptors are applied, which is exactly what happens in our example thanks to the equality $\alpha_1(x_1, x_2, x_3) = \alpha_1(x_1, x_2, \perp)$.

As for PALS, the practical upshot is a huge state space reduction, allowing model checking analysis of systems which could not be model checked in their asynchronous version because of the state space explosion caused by their concurrency. The effectiveness of this state space reduction for Multirate PALS has been demonstrated by a nontrivial case study involving a multirate distributed system controlling the turning maneuver of an airplane, where the pilot gives commands to the system, which controls the ailerons in the two wings and the rudder in the tail, all operating at different, faster rates than that of the central controller [10].

6.2.5. Advantages and shortcomings

Two obvious advantages are the great reduction in the system complexity of a multirate DRTS for design, verification, and implementation purposes, and the modularity and genericity of the pattern. A subtle point, hinted at in the above discussion, is that the bounds Γ, the period T, and the adaptors α co-determine each other. This means that even the much simpler synchronous design $\mathcal{E}$ must still take into account what can be feasibly implemented given the underlying infrastructure.

7. Related work and conclusions

7.1. Related work

There has been extensive work on patterns at the informal level; see, e.g., [30,16,50]. There has also been extensive work on formalizing patterns, e.g., [5,44,49,28,55,52,22,2]. Comparing with that work, it seems fair to say that these are valuable contributions but: (i) my focus on distributed systems makes new contributions; and (ii) except for [49], where LOTOS is

used, the formalizations in the work just cited are typically not executable, so that they should be viewed more as formal requirements than as algorithmic solutions. For example, the behavioral aspects of patterns in [44,55,22] are formalized in Lamport's TLA [35], or extensions of it such as DisCo [44]. By contrast, as pointed out in footnote 1, I consider the use of *executable* specifications in a computational logic, as opposed to just giving axioms stating formal requirements, essential to capture design patterns as *computational* solutions.

The assume-guarantee aspects of a pattern; that is, the idea that its formal guarantees can be viewed as a *contract* [42] that will be kept if the pattern is instantiated in a way that respects its semantic requirements, is emphasized for example in [52], where, in what appears to be a violation of C.S. Peirces's ethics of terminology, "assume/guarantee" is renamed to "responsibilities/rewards." Since my own work is on formal patterns for *distributed* systems, it is of course also related to the extensive literature on assume/guarantee reasoning for concurrent systems, e.g., [18,1,46,57].

As already pointed out, the work presented here on formal patterns for *distributed* systems is closely related to work on distributed object reflection [3,21,41,56]. Also, my earlier joint work with Lui Sha [51] on formal patterns for cyber-physical systems was a sketch of things to come and is in the same spirit as the ideas presented here. The already-cited work on which the formal patterns summarized in this paper is based [54,45,39,17,34,9,6,27] has been very valuable in developing the more general ideas presented here.

7.2. Conclusions and future work

Distributed systems, possibly with real-time and probabilistic features, which can *adapt* to changing environment and performance demands and come with strong formal guarantees about their behavior are quite *complex*, and hard to design, verify and build. In this paper I have proposed *formal patterns*, specified as theory transformations in rewriting logic, as a way to *drastically reduce all aspects of system complexity* when designing, verifying, implementing, and maintaining systems of this kind. The general idea is that many practical systems can then be developed by *composing* such patterns, greatly reducing system complexity and design, verification, and implementation costs, and greatly increasing their quality.

I have illustrated the proposed notion of formal pattern and its usefulness with a variety of *example patterns*. These examples show in practice that all features (1)–(6) needed for having an adequate semantic framework in which to express formal patterns (see Section 1.1) are well supported by rewriting logic.

Of course much work remains to be done. For example, although rewriting logic specifications can be directly used to derive distributed system implementations (see, e.g., [19,7,53]), I have not addressed the important issue of deriving correct-by-construction implementations of formal patterns in conventional languages. This is an important topic for future research.

Another important area of future work is augmenting the *mathematical proofs* of a formal pattern's properties with machine-assisted proofs. This is a labor-intensive process, but very much worth doing, since it further increases the high confidence that can be placed on a pattern. Likewise, machine-assisted proofs for such properties at the conventional code level for code generators and code weavers that compose the input system's code with the pattern's code would be nontrivial to obtain but extremely valuable. Proof efforts of this kind can be amortized over a large number of pattern instantiations.

The developing of *libraries* of formal patterns which contain formal executable specifications, correctness proofs, and correct-by-construction conventional code implementations to solve many distributed system problems is an even more ambitious long-term objective.

Acknowledgements

As documented in the references cited, the different formal patterns that I will use to ground the main ideas of this paper are *joint work* with the following colleagues:

- *Parameterized Modules* in Maude are joint work with Francisco Durán, Steven Eker, and other members of the Maude team.
- The *Command Shaper* pattern is joint work with Mu Sun and Lui Sha at UIUC.
- The formalization of *PALS* is joint work with Peter Ölveczky at University of Oslo; and PALS itself with Steve Miller and Darren Cofer at Rockwell-Collins, Lui Sha, Abdullah Al-Nayeem and Mu Sun at UIUC, and Peter Ölveczky at University of Oslo.
- The *Cookies Wrappers* are joint work with Rohit Chadha, Carl Gunter, Ravinder Shankesi and Mahesh Viswanathan at UIUC.
- The rewriting logic formalization of *ASV Wrappers* is joint work with Musab AlTurki and Carl Gunter at UIUC.
- The *Server Replicator Wrapper*, and its composition with an improved version of the ASV wrappers are joint work with Musab AlTurki at UIUC, Jonas Eckhardt and Tobias Mühlbauer at TU Munich, and Martin Wirsing at LMU Munich.
- The *Multirate PALS* pattern, and its auxiliary *k-Step Machine* and *Input Adaptor* patterns, are joint work with Kyungmin Bae at UIUC and Peter Ölveczky at University of Oslo.

I thank the organizers of FACS 2011 for giving me the opportunity of presenting these ideas in Oslo, and all the participants for their interest and comments. I thank the anonymous referees for their constructive criticism and their suggestions,

which have all helped in improving the paper. Kyungmin Bae, Jonas Eckhardt, Tobias Mühlbauer, Camilo Rocha and Mu Sun deserve special thanks for their help with various pictures and diagrams in the paper. This work has been supported in part by the Boeing Corporation under Grant C8088-557395, NSF Grants CNS 08-34709, CCF 09-05584, and AFOSR Grant FA8750-11-2-0084.

References

[1] M. Abadi, L. Lamport, Composing specifications, ACM Trans. Program. Lang. Syst. 15 (1) (1993) 73–132.
[2] J.-R. Abrial, T.S. Hoang, Using design patterns in formal methods: an event-B approach, in: J.S. Fitzgerald, A.E. Haxthausen, H. Yenigün (Eds.), ICTAC, in: LNCS, vol. 5160, Springer, 2008, pp. 1–2.
[3] G. Agha, S. Frolund, R. Panwar, D. Sturman, A linguistic framework for dynamic composition of dependability protocols, IEEE Parallel Distrib. Technol. 1 (1993) 3–14.
[4] G. Agha, J. Meseguer, K. Sen, PMaude: Rewrite-based specification language for probabilistic object systems, Electron. Notes Theor. Comput. Sci. 153 (2) (2006) 213–239.
[5] P.S.C. Alencar, D.D. Cowan, C.J.P. de Lucena, A formal approach to architectural design patterns, in: M.-C. Gaudel, J. Woodcock (Eds.), FME, in: LNCS, vol. 1051, Springer, 1996, pp. 576–594.
[6] M. AlTurki, Rewriting-based formal modeling, analysis and implementation of real-time distributed services, PhD thesis, University of Illinois at Urbana-Champaign, 2011, http://hdl.handle.net/2142/26231.
[7] M. AlTurki, J. Meseguer, Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis, in: P.C. Ölveczky (Ed.), Proc. 1st Intl. Workshop on Rewriting Techniques for Real-Time Systems, RTRTS 2010, in: Electron. Proc. Theor. Comput. Sci., vol. 36, 2010, pp. 26–45.
[8] M. AlTurki, J. Meseguer, PVeStA: A parallel statistical model-checking and quantitative analysis tool, in: Proc. CALCO 2011, in: LNCS, vol. 6859, Springer, 2011, pp. 386–392.
[9] M. AlTurki, J. Meseguer, C. Gunter, Probabilistic modeling and analysis of DoS protection for the ASV protocol, Electron. Notes Theor. Comput. Sci. 234 (2009) 3–18.
[10] K. Bae, J. Krisiloff, J. Meseguer, P.C. Ölveczky, PALS-based analysis of an airplane multirate control system in Real-Time Maude, in: P.C. Ölveczky, C. Artho (Eds.), FTSCS, in: EPTCS, vol. 105, 2012, pp. 5–21.
[11] K. Bae, J. Meseguer, P.C. Ölveczky, Formal patterns for multi-rate distributed real-time systems, in: C.S. Pasareanu, G. Salaün (Eds.), FACS, in: LNCS, vol. 7684, Springer, 2012, pp. 1–18.
[12] J. Bergstra, J. Tucker, Characterization of computable data types by means of a finite equational specification method, in: J.W. de Bakker, J. van Leeuwen (Eds.), Automata, Languages and Programming, Seventh Colloquium, in: LNCS, vol. 81, Springer-Verlag, 1980, pp. 76–90.
[13] D. Bernstein, Syn cookies, http://cr.yp.to/syncookies.html, 1996.
[14] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, ELAN from a rewriting logic point of view, Theor. Comput. Sci. 285 (2002) 155–185.
[15] R. Burstall, J.A. Goguen, The semantics of Clear, a specification language, in: D. Bjorner (Ed.), Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification, in: LNCS, vol. 86, Springer, 1980, pp. 292–332.
[16] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, Pattern-Oriented Software Architecture, vol. 1: A System of Patterns, Addison–Wesley, 1996.
[17] R. Chadha, C.A. Gunter, J. Meseguer, R. Shankesi, M. Viswanathan, Modular preservation of safety properties by cookie-based DoS-protection wrappers, in: Proc. FMOODS 2008, in: LNCS, vol. 5051, Springer, 2008, pp. 39–58.
[18] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison–Wesley, 1988.
[19] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, C. Talcott, All About Maude – A High-Performance Logical Framework, LNCS, vol. 4350, Springer, 2007.
[20] M. Clavel, J. Meseguer, M. Palomino, Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic, Theor. Comput. Sci. 373 (2007) 70–91.
[21] G. Denker, J. Meseguer, C. Talcott, Rewriting semantics of meta-objects and composable distributed services, in: Proc. 3rd Intl. Workshop on Rewriting Logic and Its Applications, in: ENTCS, vol. 36, Elsevier, 2000.
[22] J. Dong, P.S.C. Alencar, D.D. Cowan, S. Yang, Composing pattern-based components and verifying correctness, J. Syst. Softw. 80 (11) (2007) 1755–1769.
[23] F. Durán, S. Eker, P. Lincoln, J. Meseguer, Principles of Mobile Maude, in: Agent Systems, Mobile Agents, and Applications, ASA/MA 2000, in: Springer LNCS, vol. 1882, Springer-Verlag, 2000, pp. 73–85.
[24] F. Durán, J. Meseguer, Structured theories and institutions, Theor. Comput. Sci. 309 (1–3) (2003) 357–380.
[25] F. Durán, J. Meseguer, Maude's module algebra, Sci. Comput. Program. 66 (2) (2007) 125–153.
[26] J. Eckhardt, A formal analysis of security properties in cloud computing, Master's thesis, LMU, Munich, 2011.
[27] J. Eckhardt, T. Mühlbauer, M. AlTurki, J. Meseguer, M. Wirsing, Stable availability under denial of service attacks through formal patterns, in: Lara de Zisman (Ed.), FASE, in: LNCS, vol. 7212, Springer, 2012, pp. 78–93.
[28] A.H. Eden, Y. Hirshfeld, Principles in formal specification of object oriented design and architecture, in: D.A. Stewart, J.H. Johnson (Eds.), CASCON, IBM, 2001, p. 3.
[29] K. Futatsugi, R. Diaconescu, CafeOBJ Report, AMAST Series, World Scientific, 1998.
[30] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, John Wiley & Sons, 1994.
[31] J. Goguen, R. Burstall, Institutions: Abstract model theory for specification and programming, J. ACM 39 (1) (1992) 95–146.
[32] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud, Introducing OBJ, in: Software Engineering with OBJ: Algebraic Specification in Action, Kluwer, 2000, pp. 3–167.
[33] C.A. Gunter, S. Khanna, K. Tan, S.S. Venkatesh, DoS protection for reliably authenticated broadcast, in: Network and Distributed System Security, NDSS 04, The Internet Society, 2004.
[34] S. Khanna, S.S. Venkatesh, O. Fatemieh, F. Khan, C.A. Gunter, Adaptive selective verification, in: INFOCOM, IEEE Press, 2008, pp. 529–537.
[35] L. Lamport, A temporal logic of actions, ACM Trans. Program. Lang. Syst. 16 (3) (1994) 872–923.
[36] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theor. Comput. Sci. 96 (1) (1992) 73–155.
[37] J. Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G. Agha, P. Wegner, A. Yonezawa (Eds.), Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993, pp. 314–390.
[38] J. Meseguer, Membership algebra as a logical framework for equational specification, in: F. Parisi-Presicce (Ed.), Proc. WADT'97, in: LNCS, vol. 1376, Springer, 1998, pp. 18–61.
[39] J. Meseguer, P.C. Ölveczky, Formalization and correctness of the PALS architectural pattern for real-time systems, in: 12th International Conference on Formal Engineering Methods, ICFEM 2010, in: LNCS, vol. 6447, Springer, 2010, pp. 303–320.
[40] J. Meseguer, P.C. Ölveczky, Formalization and correctness of the PALS architectural pattern for distributed real-time systems, Theor. Comput. Sci. 451 (2012) 1–37.

[41] J. Meseguer, C. Talcott, Semantic models for distributed object reflection, in: Proceedings of ECOOP'02, Málaga, Spain, in: LNCS, vol. 2374, Springer, June 2002, pp. 1–36.
[42] B. Meyer, Object-Oriented Software Construction, Prentice Hall, 1997.
[43] B. Meyer, K. Arnout, Componentization: The visitor example, IEEE Comput. 39 (7) (2006) 23–30.
[44] T. Mikkonen, Formalizing design patterns, in: ICSE, 1998, pp. 115–124.
[45] S. Miller, D. Cofer, L. Sha, J. Meseguer, A. Al-Nayeem, Implementing logical synchrony in integrated modular avionics, in: Proc. 28th Digital Avionics Systems Conference, IEEE, 2009.
[46] J. Misra, A Discipline of Multiprogramming, Springer-Verlag, 2001.
[47] T. Mühlbauer, Formal specification and analysis of cloud computing management, Master's thesis, LMU, Munich, 2011.
[48] P.C. Ölveczky, J. Meseguer, Semantics and pragmatics of Real-Time Maude, High.-Order Symb. Comput. 20 (1–2) (2007) 161–196.
[49] M. Saeki, Behavioral specification of GOF design patterns with LOTOS, in: APSEC, IEEE Computer Society, 2000, pp. 408–415.
[50] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture, vol. 2: Patterns for Concurrent and Networked Objects, John Wiley & Sons, 2000.
[51] L. Sha, J. Meseguer, Design of complex cyber physical systems with formalized architectural patterns, in: Software-Intensive Systems and New Computing Paradigms, in: LNCS, vol. 5380, Springer, 2008, pp. 92–100.
[52] N. Soundarajan, J.O. Hallstrom, Responsibilities and rewards: Specifying design patterns, in: ICSE, IEEE Computer Society, 2004, pp. 666–675.
[53] M. Sun, J. Meseguer, Distributed real-time emulation of formally-defined patterns for safe medical device control, in: RTRTS'10, in: EPTCS, vol. 36, 2010, pp. 158–177.
[54] M. Sun, J. Meseguer, L. Sha, A formal pattern architecture for safe medical systems, in: Proc. WRLA 2010, in: LNCS, vol. 6381, Springer, 2010, pp. 157–173.
[55] T. Taibi, D.N.C. Ling, Formal specification of design patterns – a balanced approach, J. Object Technol. 2 (4) (2003) 127–140.
[56] N. Venkatasubramanian, C.L. Talcott, G. Agha, A formal model for reasoning about adaptive QoS-enabled middleware, ACM Trans. Softw. Eng. Methodol. 13 (1) (2004) 86–147.
[57] M. Viswanathan, R. Viswanathan, Foundations for circular compositional reasoning, in: Proc. ICALP'01, in: LNCS, vol. 2076, Springer, 2001, pp. 835–847.