Role-Based Declarative Synchronization for Reconfigurable Systems*

Vlad Tanasescu and Paweł T. Wojciechowski

Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
{Vlad.Tanasescu,Pawel.Wojciechowski}@epfl.ch

Abstract. In this paper we address the problem of encoding complex concurrency control in reconfigurable systems. Such systems can be often reconfigured, either statically, or dynamically, in order to adapt to new requirements and a changing environment. We therefore take a declarative approach and introduce a set of high-level programming abstractions which allow the programmer to easily express complex synchronization constraints in multithreaded programs. The constructs are based on our model of *role-based synchronization (RBS)* which assumes attaching roles to concurrent threads and expressing a synchronization policy between the roles. The model is illustrated by describing an experimental implementation of our language as a design pattern library in OCaml. Finally, we also sketch a small application of a web access server that we have implemented using the RBS design pattern.

1 Introduction

Our motivating example of reconfigurable systems are networked applications, such as modular group communication middleware protocols [7, 14] and web services [27]. Software components in these applications interact with clients and process network messages. Due to efficiency reasons, different components (or objects) may be accessed concurrently, with possible dependencies on other components. To provide data consistency and a quality of service, complex synchronization policies of object accesses are required. Unfortunately, a given composition of application components may often change in order to adapt to a new environment or changing user requirements. This means that the synchronization policy may need to be revised as well, and the corresponding code changed accordingly, making programming of such systems a difficult task.

Developing multithreaded systems is considerably more difficult than implementing sequential programs due to several reasons:

– traditional concurrency constructs, such as monitors and conditional variables, are used to express synchronization constraints at the very low level of individual accesses to shared objects (thread safety);

* Research supported by the Swiss National Science Foundation under grant number 21-67715.02 and Hasler Stiftung under grant number DICS-1825.

- embedding the implementation of a synchronization policy in the main code compromises both a good understanding of the application logic, i.e. we are not sure from the first look what the application code does, and also an understanding of the policy expressed;
- the notions of *thread roles* such as producer or consumer, which are essential for the understanding of a given policy, tend to disappear beyond an accumulation of lines of code, just as the logical essence of a sequential program gets lost when expressed in, say, an assembly language;
- synchronization constructs are usually entangled with instructions of the main program, which means that the correctness of concurrent behaviour is bound to the correctness of the entire application; this feature complicates maintenance and code reuse – some of the most advocated reasons for using components.

We therefore study *declarative synchronization*, which assumes a separation of an object's functional behaviour and the synchronization constraints imposed on it. Such an approach enables to modify and customize synchronization policies constraining the execution of concurrent components, without changing the code of component objects, thus making programming easier and less error-prone.

While some work on such separation of concerns exists (see [8, 5, 23, 22, 16] among others) and example languages have been built (see [21, 22, 15, 16]), as far as we know, our solution is novel. It shares a number of design features with these languages, such as support for "declarative style" and "separation of synchronization aspects". However, there are also important differences in the model and implementation (we characterize them briefly in §2). Some of our motivations and goals are different, too.

In this paper, we propose a *role-based synchronization (RBS)* model with a constraint language to express concurrency control between the roles. Our design has been guided by two main requirements:

- to keep the semantics of synchronization control attached to roles involved in the specification of a concurrent problem rather than to instances of components and objects (or their abstractions);
- to allow expressing concurrent strategies independently from the main code of applications, with a possibility to switch between different strategies on-the-fly (with some control on the moment of switching).

Our long term goal is to develop support of declarative synchronization for component-based systems that can be dynamically reconfigured or adapted to changing requirements or a new environment. For instance, when a mobile device is moved from outside to inside a building, it may reconfigure its suite of network protocols on-the-fly. Another example are mobile collaborative systems, such as the *Personal Assistant (PA)* application [29]; the PA service components may need to switch between user interfaces at runtime, depending on a given device accessed by the user at a time (e.g. a hand-held device or PC workstation).

In our previous work, Wojciechowski [28] has studied typing and verification of synchronization policies expressed using *concurrency combinators* – a simple language for declaring constraints between static modules (instead of dynamic roles). This paper provides an alternative model and new insight into the implementation aspects of declarative synchronization.

To illustrate our approach, we describe an example RBS design pattern package that we have implemented in OCaml [17]. Notably, our experimental implementation tackles both aspects of our design (i.e. separation of concerns and expressiveness) without using any precompilation tools. We believe however that more traditional programming languages, such as Java [6] and C++ [26], could be also used.

Our current implementation of RBS can only switch between synchronization policies that have been predefined by a given RBS package. This is sufficient for the above example applications of reconfigurable systems. Ultimately, we would like to be able to download a new policy dynamically. The OCaml distribution does not support dynamic class loading, however. We intend therefore to experiment with programming languages that support dynamic data types with dynamic typing and binding, such as Python [20] and Acute [24]. The latter is an extension of the OCaml language with dynamic loading and controlled rebinding to local resources. We leave this for future work.

The paper is organized as follows. §2 contains related work, §3 introduces the RBS model and constraint language, §4 describes an example RBS package, §5 discusses dynamic switching between synchronization policies, §6 illustrates our approach using a small web application, and §7 concludes.

2 Related Work

There have been recently many proposals of concurrent languages with novel synchronization primitives, e.g. Polyphonic C# [1] and JoCaml [3] that are based on the join-pattern abstraction [4], and Concurrent Haskell [10], Concurrent ML [18], Pict [19] and Nomadic Pict [25, 29], with synchronization constructs based on channel abstractions. They enable to encode complex concurrency control more easily than when using standard constructs, such as monitors and locks.

The above work is orthogonal to the goals of this paper. We are primarily focused on a declarative way of encoding synchronization through separation of concerns (see [8, 12, 11, 9] among others). The low-level details of the RBS implementation resemble the idea of aspect-oriented programming. Below we discuss example work in these two areas.

Separation of Concurrency Aspects. For a long time, the object-oriented community has been pointing out, under the term *inheritance anomaly* [13], that concurrency control code interwoven with the application code of classes, can represent a serious obstacle to class inheritance, even in very simple situations.

For instance, consider a library class `SimpleBuffer` implementing a bounded buffer shared between concurrent producers and consumers. The class provides public methods `input()` and `output()` which can be used to access the buffer. The implementation of these methods would normally use some conditional variables like `is_buffer_full` or `is_buffer_empty` in order to prevent the buffer from being accessed when it is full or empty. Suppose we want to implement a buffer in which we would like to add a condition that nobody can access the buffer after a call to a method `freezeBuffer()` has been made. But in this case, we are not able to simply extend the class `SimpleBuffer`. We also need to rewrite the code of both methods `output()` and `input()` in order to add the new constraint on the buffer usage!

Milicia and Sassone [15, 16] address the inheritance anomaly problem and propose an extension of Java with a linear temporal logic to express synchronization constraints on method calls. Their approach is similar to ours (although our motivation is the ease of programming and expressiveness). However, it requires a precompilation tool in order to translate a program with temporal logic clauses into Java source code, while our approach uses the facilities provided by the host language. Also, their language does not allow for expressing synchronization constraints that require access to a program's data structures.

Ramirez *et al.* [21, 22] have earlier proposed a simple constraint logic language for expressing temporal constraints between "marked points" of concurrent programs. The approach has been demonstrated using Java, extended with syntax for marking. Similarly to the approach in [15, 16], the language has however limited expressiveness. While our constraint declarations can freely access data in a thread-safe way, and call functions of the application program, their constraints are not allowed to refer to program variables. Also, composite synchronization policies (on groups of threads) are not easily expressible.

The previous work, which set up goals similar to our own is also by Ren and Agha [23] on separation of an object's functional behaviour and the timing constraints imposed on it. They proposed an actor-based language for specifying and enforcing at runtime real-time relations between events in a distributed system. Their work builds on the earlier work of Frølund and Agha [5] who developed language support for specifying multi-object coordination, expressed in the form of constraints that restrict invocation of a group of objects.

In our previous work, Wojciechowski [28] has studied declarative synchronization in the context of a calculus of concurrency combinators. While in this paper we propose a language for expressing dynamic constraints between role-oriented *threads*, the concurrency combinators language is used to declare synchronization constraints between static *code fragments*. The calculus is therefore equipped with a static type system that can verify if the matching of a policy and program is correct. Typable programs are guaranteed to make progress.

Aspect-Oriented Programming. *Aspect-Oriented Programming (AOP)* is a new trend in software engineering. The approach is based on separately specifying the various *concerns* (or *aspects*) of a program and some description of their relationship, and then relying on the AOP framework to *weave* [9] or compose them together into a coherent program. For instance, error handling or security checks can be separated from a program's functional core. Hürsch and Lopes [8] identify various concerns, including synchronization. Lopes [12] describes a programming language D, that allows thread synchronization to be expressed as a separate concern. More recently, AOP tools have been proposed for Java, such as AspectJ [11]. They allow aspects to be encoded using traditional languages and weaved at the intermediate level of Java bytecode. The programmer writes aspect code to be executed *before* and *after* the execution of *pointcuts*, where a pointcut usually corresponds to invocations of an application method.

The code weaving techniques can be, of course, applied to synchronization aspects too, and by doing so, we can achieve separation of concurrency concerns. However, by using a pure AOP approach, we are not getting yet more expressiveness. In this paper, we propose a set of language abstractions that are used to *declare* an arbitrary synchronization policy. The policy is then effectuated automatically at runtime by a concurrency controller implementing a given RBS design pattern. Our current implementation of RBS design patterns in OCaml does not resort to external precompilation tools.

In §3, we describe the RBS model and the constraint language. Then we explain the implementation details in §4, using an example RBS design pattern.

3 Role-based Synchronization

In this section, we describe our simple but expressive model of Role-Based Synchronization (RBS). By looking at the classical concurrency problems, such as Producer-Consumer, Readers-Writers, and Dining Philosophers, we can identify two essential semantic categories which are used to describe these problems, i.e. roles and constraints imposed on the roles. Below we characterize these two categories.

3.1 Thread Roles

Threads in concurrent programs are spawned to perform certain *roles*, e.g. producers, consumers, readers, writers, and philosophers. Below we confuse roles and threads unless otherwise stated, i.e. a role means one, or possibly many concurrent threads, that are logically representing the role.

Roles can execute *actions*, e.g. to output a value in the buffer, to write a value to a file, to eat rice. Roles can be in different *states* during program execution. Some actions are allowed only in certain states, i.e. in order to execute an action a role must first *enter* a state that allows the action to be executed, unless the role is already in such a state.

The common synchronization problems are usually concerned with accessing *shared resources* (or *objects*) by roles in an exclusive manner, e.g. a buffer, a file, a fork and a rice bowl. We can therefore identify two role states (more refined states can also be defined): the state of being able to call methods of a shared object (and execute the corresponding action) and the state of waiting to be able to do so. We denote by `In` the former state (where "in" means being *in* the position to execute actions) and by `Wait` the latter state.

For instance, a producer is waiting if the buffer is full, a writer is waiting when another writer is writing, a philosopher is waiting if at least one fork is missing. Otherwise, these roles are in the position to execute all role's actions which have been defined on the buffer, file, and rice bowl.

3.2 Synchronization Constraints

Synchronization constraints describe conditions on when roles are allowed to enter a state and execute the role's actions allowed by the state.

For instance, a consumer can only input a value if neither producer nor other consumer is currently accessing the buffer *and* there is actually some value in the buffer to be obtained; many concurrent readers can read a file concurrently if there is no writer writing to the file at the same time; and a philosopher can eat rice only if two forks are available.

Thus, synchronization constraints specify a synchronization problem – if an implementation obeys all the constraints defined, then it provides a correct solution to the problem. Failure to satisfy any constraint in accessing shared objects by roles may cause incorrect program behaviour. Below we describe two parts that jointly define a synchronization constraint: a synchronization policy and a logical condition.

Synchronization Policy. A *synchronization policy* (or *policy* in short) defines a constraint between different roles and their states. We say that a policy is *satisfied* if the constraint always holds, and it is *violated* otherwise. Essentially, a policy specifies when a role is permitted or forbidden to enter a given state.

A popular policy is "mutual exclusion", which simply states that some roles cannot be simultaneously in the critical section defined by access to a shared object (method, data field, etc.). For instance, a producer and consumer cannot access a buffer at the same time, two writers are not allowed to simultaneously write to the file, and two philosophers cannot share the same fork. More precisely, they are not allowed at any time to be both in the same state `In`.

Logical Conditions. Satisfying the synchronization policy is the necessary but often not sufficient condition to solve a synchronization problem. For instance, a consumer cannot input a value if there is no value in the buffer. Thus, we also introduce a *logical condition* which specifies a constraint on an object requested by a role; this condition must be satisfied in order to allow a method of the object to be called by the role.

3.3 Constraint Language

The above observations led us to a simple constraint language, defined in Fig. 1. The language is expressive enough to describe most of the common concurrency problems. Below we describe the syntactic constructs and example policy types. We use $[\overline{x}]$ to denote a list of elements $\overline{x} = x_1, ..., x_n$, where $[]$ is the empty list.

Objects. Shared objects, denoted o, are declared as lists of pairs $(a, [\overline{S}])$, where the meaning of each pair is that the object method a (action a) can be called (executed) by a role only if the role is in one of the states mentioned in the list $[\overline{S}]$ (an empty list $[]$ is used if the method can be called in *any* state).

For instance, `Buffer = [(output, [In]); (is_full, [])]` declares an object `Buffer` which has a method `output`, which can be called only by a role which is in state `In`, and a method `is_full` which can be called in any state.

Constraint Language:

Thread roles	$R \in Roles$		$o ::= [(a, [\overline{S}]); ...; (a, [\overline{S}])]$
Objects	$o \in Objects$		$F ::= ([\overline{R}], [\overline{K}])$
Actions	$a \in Actions$		$K ::= \texttt{enter}(R, S) = (P, C)$
States	$S \in States$		$P ::= (T, [(S, [\overline{R}]); ...; (S, [\overline{R}])])$
Families	$F \in Families$		$T ::= \texttt{Excluded} \mid \texttt{Allowed} \mid \texttt{Required} \mid ...$
Constraints	$K \in Constraints$		
Sync policies	$P \in Policies$		
Policy types	$T \in Types$		
Conditions	$C \in o.a \rightarrow boolean$		

Example Policy Types:

$$\frac{\exists R' \in \overline{R}_i.\ R' \text{ in } S_i \quad \text{for all} \quad i = 1..n}{(\texttt{Allowed}, [(S_1, [\overline{R}_1]); ...; (S_n, [\overline{R}_n])]) \quad \text{satisfied}}$$

$$\frac{\forall R' \in \overline{R}_i.\ R' \text{ in } S_i \quad \text{for all} \quad i = 1..n}{(\texttt{Required}, [(S_1, [\overline{R}_1]); ...; (S_n, [\overline{R}_n])]) \quad \text{satisfied}}$$

$$\frac{\forall R' \in \overline{R}_i.\ R' \text{ not in } S_i \quad \text{for all} \quad i = 1..n}{(\texttt{Excluded}, [(S_1, [\overline{R}_1]); ...; (S_n, [\overline{R}_n])]) \quad \text{satisfied}}$$

Fig. 1. The Role-Based Synchronization Model

Constraints and Families. We define a *constraint* on entering a state S by a role R, written $\texttt{enter}(R, S)$, to be a policy P which regulates switching of the role to this state, paired with a logical condition C on all objects accessible by the role in state S. The role R can enter the state S if the policy P is satisfied *and* the condition C is true.

Policy P is expressed as a policy type T paired with a list L of *policy rules*, i.e. tuples $(S, [\overline{R}])$ of a state and a list of roles; the meaning of a policy rule will be explained below. The condition C has the form of a boolean function with no arguments which returns **true** if calling methods by role R will actually make sense if role R would now enter the state S, and **false** otherwise. What "makes sense" or not depends, of course, on the program semantics. Programmers define function C as part of the main code of the application, and so any variables visible in the scope of the function can be used to express the condition.

We define a *role family*, denoted F, to be a list of roles of a single synchronization problem, paired with a list of constraints, e.g. `Produce-Consumer-Family = ([Producer; Consumer], [enter(Producer, In); enter(Consumer, In)])`. Roles are globally unique.

Policy Types. We have identified two categories of policies: permission and denial (or refusal). Intuitively, a *permission policy* describes what must happen in order to *permit* a role to enter a state, while a *denial policy* describes what *forbids* a role to enter a state. Two example permission policy types and one denial policy type have been defined in the bottom of Figure 1:

Consider a constraint $\texttt{enter}(R, S) = (P, C)$, where $P = (T, L)$. We have the following policy types T:

- The $T =$ `Allowed` policy says that role R can enter state S only if for each tuple $(S_i, [\overline{R}_i])$ in L, at least one role in $\overline{R}_i$ is in state S_i; the empty list of roles means *any* role.
- The $T =$ `Required` policy says that role R can enter state S only if for each tuple $(S_i, [\overline{R}_i])$ in L, all roles $\overline{R}_i$ are in state S_i; the empty list of roles means *all* roles.
- The $T =$ `Excluded` policy says that role R can enter state S only if for each tuple $(S_i, [\overline{R}_i])$ in L, all roles in $\overline{R}_i$ are *not* in state S_i; the empty list of roles means *all* roles.

Note that satisfying policy `Excluded` means that policy `Allowed` is violated (and vice versa); similarly we can define the fourth denial policy by negation of `Required`.

Example Specifications of Constraints. For instance, consider a Producer-Consumer problem with the priority of producers. The policy of accessing a buffer by a consumer is such that the consumer is forbidden to enter a state that allows the buffer to be accessed, if there is already a producer accessing the buffer or there are some producers waiting in the queue to access it. Moreover, a consumer can enter this state only when the buffer is not empty.

We can specify the above constraint using an exclusion policy and a logical condition, as follows: `enter(Consumer, In) = (Excluded, [(In, [Producer]), (Wait, [Producer])], not (buffer.empty))`.

On the other hand, a constraint on entering the state `In` by producer is: `enter(Producer, In) = (Excluded, [(In, [Consumer])], not (buffer.full))`.

The simplicity of this formalism suggests that it can be indeed useful to encode synchronization constraints at the level of thread roles, instead of individual actions executed by the threads. The advantage is that the constraints can be expressed *declaratively*, as a set of policy rules. The rules are intuitively easier to understand than the low-level synchronization code, thus aiding design and proofs of correctness.

In the next section, we demonstrate how the separation of concerns defined by our model can be achieved in practice.

4 Example 'RBS' Package

We illustrate our approach using an example role-based synchronization package that we have implemented in the OCaml programming language [17] – an object-oriented variant of ML. OCaml has abstract types and pattern matching over types, which allowed us to use the syntax of the constraint language exactly as it has been defined in §3.3. We think however that the approach described in this paper can be easily adapted to any object-oriented language.

The package *Readers-Writers (RW)* implements a role family of the Readers-Writers synchronization problem [2], defined as follows. Two kinds of threads – readers and writers – share an object. To preclude interference between readers and writers, a writer must have exclusive access to the object. Assuming no writer is accessing the object, any number of readers may concurrently access the object. We assume that writers have priority over readers, i.e. new readers are delayed if a writer is writing or waiting, and a delayed reader is awakened only if no writer is waiting. To demonstrate expressiveness of our language, we have slightly extended the problem by adding a new role `Sysadmin` that represents a system administrator. The `Sysadmin` role is to periodically turn the computer – i.e., the shared object in our example – off and on for maintenance; turning the computer off prevents both writers and readers from executing their actions.

The example code below is taken almost verbatim from the source code in OCaml. However, those readers who are familiar with any object-oriented language should not find the programs too difficult to understand.

Application Program. Imagine a programmer who wants to use the RW package to implement an application that requires the Readers-Writers synchronization pattern. We assume that the programmer has already defined all application classes but no synchronization code is provided yet. Below is an example class:

```
class computer =
  object
    val mutable screen = 0
    val mutable on = true
    method read = screen
    method write x = screen <- x    (* Assign a new value to 'screen' *)
    method is_working = on
    method turn = on <- (not on)    (* Invert the boolean value 'on' *)
  end
```

The class implements a "computer object" that will be accessed by concurrent readers and writers. The class defines methods `read` and `write` for accessing the object's shared state `screen`, and methods `turn` and `is_working` that are used to turn the computer on/off and check its current status (on/off).

To use the RW synchronization package, the programmer has to create instances of *role objects*, where each role object (`r`, `w`, `s`) represents a particular role and "wraps" method calls on objects accessible by the role (in our case it is the computer object only):

```
let o = new computer;
let r = new rw_Reader o;
let w = new rw_Writer o;
let s = new rw_Sysadmin o;
```

Now we just require that concurrent threads (roles) in the application program do not access shared objects directly, but via the role objects defined above. For instance, a thread representing role `Reader` must read on the (screen of) computer using a method `r.read` instead of `o.read`, similarly for other roles.

Below is an example code which creates a thread of role `Writer` that writes in a loop a random integer (the notation `w#write` in OCaml corresponds to `w.write` in languages like Java and C++, i.e. "call a method `write` of object `w`").

```
let thread_writer_no1 = Thread.create (fun () ->
  while true do
    w#write (Random.int 10)
  done)
();
```

The invocation of the method by thread `thread_writer_no1` is guaranteed to satisfy all the synchronization constraints imposed on this call, i.e. the call will be suspended (and the thread blocked) if the constraints cannot be satisfied, and automatically resumed after actions made by other threads will allow the constraints to be satisfied.

Synchronization Package. Now, let us explain how the RW package has been implemented. Below are declarations of roles and states:

```
(* Roles defined in Readers/Writers Package *)
type role = Reader | Writer | Sysadmin

(* States visited by roles *)
type action = Wait | In
```

Below is a class which implements "wrapping" of the shared object for the role `Writer` (the classes and methods have polymorphic types, expressed in OCaml with a type variable `'a`):

```
class ['a] rw_Writer o =
  object
    (* Use object 'synchronizer' to evaluate constraints *)
    inherit ['a] rbs Writer synchronizer as sync

    method write (x : 'a) : unit =
      (* Message sent to synchronizer before the call *)
      sync#try_cond ([(Begin, Wait)], [(End, Wait);
                                       (Begin, In)]);
      (* Call the actual method (only one here) *)
      let result = o#write x in
      (* Message sent to synchronizer after the call *)
      sync#notify [(End, In)];
      (* Return any method result *)
      result
  end
```

Essentially, each call of an object method by a role is preceded and followed by a call to a `synchronizer` object (which has been bound to a name `sync`). Note that this resembles the AOP weaving principle. The synchronizer is the core part of the package – it implements an evaluation engine parameterized over constraint declarations (which will be explained below).

The effect of calling `sync#try_cond` above is that the engine will evaluate constraints declared for the role `Writer`. If all constraints are satisfied then a thread calling the `write` method will enter the state `In`, and the method will be invoked on the actual object `o`. Otherwise, the thread enters the state `Wait` (and it will be awaken once possible). After the call returns, the thread leaves state `In` and enters a default state `Idle`.

The method `sync#try_cond` takes as arguments two lists of events to be triggered by the role's threads, respectively at the beginning of a critical section, and inside the critical section, where an *event* is a tuple of time delimiters (`End` and `Begin`) and the states defined by a given design pattern (in our example, these are `Wait` and `In`). Triggering events increase and decrease some counter variables of the package, that are used by the concurrency controller to enforce user-defined synchronization policies. By using this mechanism, our approach allows logical conditions in declared policies to be parameterized by the number of threads that are in a given state (as illustrated in the following example).

Below are synchronization constraints for the `Reader` and `Writer` roles predefined by the package. (Of course, the application programmer who uses the package could always extend this set and express some specialized constraints.)

```
synchronizer#define_constraint
  Reader [{
    enter  = In;
    policy = (Excluded, [(In, [Writer; Sysadmin], []);
                         (Wait, [Writer], [])]);
    check  = [(fun () -> computer#is_working)]
  }];

synchronizer#define_constraint
  Writer [{
    enter  = In;
    policy = (Allowed, [(In, [], [])]);
    check  = [(fun () -> computer#is_working)]
  }];
```

Each constraint of a role consists of fields `enter`, `policy`, and `check`, which define respectively the state to enter by the role, the policy and the condition. Otherwise the syntax matches the one that we have used in §3.3, with a small extension.

To allow policies that require a quantitative information about threads, we have added a third field in each rule (tuple) of the policy list. This field may contain a number of threads of the corresponding role from which the rule begins to apply (the default value is 1), e.g. replacing `[]` by `[2]` in the `(Wait, [Writer], [])` policy (see the first constraint) would mean that the `Reader` will be excluded only if there are at least two `Writer`s waiting.

5 Dynamic Policy Switching

Our constraint language could be used to extend the separation-of-concerns principle to systems that must be reconfigured without stopping them. For such systems, we consider a *dynamic switching* of synchronization constraints.

Each constraint is guarded by a boolean function (defined in the application program). Only a constraint whose guard has returned **true** is chosen for synchronization. If several guards have returned **true**, then the order of applying the corresponding policies depends on the RBS package. The RBS package implementation makes sure that the transition between different policy types is atomic and a new policy is correctly satisfied. We could easily extend the model in §3.3 accordingly, by requiring the second component of a constraint declaration to be a triple `(C,P,C')`, where `C` is a guard (we omitted it for clarity).

Below are two example constraints on a role `Reader` to enter a state `In`. The guard functions examine a content of a boolean variable `happy_hour`. The policy switching is "triggered" on-the-fly by the program execution state. Depending on the current value stored in `happy_hour` (returned with the ! construct that is used in OCaml to read mutable data) the synchronizer will either evaluate the first constraint, if `happy_hour` contains **true**, or the second one (otherwise).

```
synchronizer#define_constraint
  Reader [{
    enter  = In;
    guard  = [(fun () -> !happy_hour)];
    policy = (Excluded, [(In, [Writer; Sysdmin], [])]);
    check  = [(fun () -> computer#is_working)]
  };
  {
    enter  = In;
    guard  = [(fun () -> not !happy_hour)];
    policy = (Excluded, [(In, [Writer; Sysadmin], []);
                         (Wait, [Writer], [])]);
    check  = [(fun () -> computer#is_working)]
  }];
```

The policy in the first constraint removes the clause about the priority of writers from the second constraint (explained in §4) and so it equals the rights of readers and writers to access the computer's screen during a "happy hour".

6 Example Application

To facilitate experimentation, we have prototyped a small web access server that provides access to remote services to a large number of clients. The clients may have quite different workload profiles and priorities, simultaneous access of some client types can be forbidden. Accessing a distant resource such as a remote web service or database can be both expensive and time consuming. Therefore such accesses should be well controlled and fine-tuned, considering different client

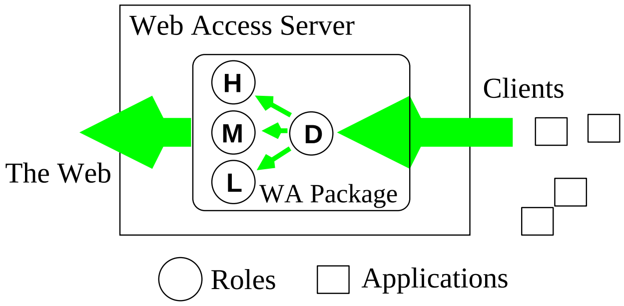

Fig. 2. The Web Access Server

profiles, and external factors such as the current traffic, the time of the day, etc. We also require that:

– to meet any future expectations without reverse-engineering the server's components, the code must be flexible and easy to maintain and modify;
– it should be possible to change the control policy dynamically.

Our solution is to use a generic *Web Access (WA)* synchronization package to implement a component that is used to access the web. We present a schema showing the main components of our implementation in Figure 2. Roles predefined by the WA package are the `Low` (`L`), `Medium` (`M`), and `High` (`H`) *access priority* roles and the `Dispatcher` role `D`.

The access priority roles define different policies of accessing the web based on the client's priority (low, medium, high). The general policy constraint *between* access priority roles is that roles `M` and `L` are blocked if role `H` is accessing the web, while `L` is blocked if either `M` or `L` or both are accessing the web. Internally, each priority role may declare several variants of the policy which may change dynamically, e.g. once moving from the peak time of the day to the evening, or when some traffic fluctuation has been observed.

The `Dispatcher` role recognizes a client contacting the server and dynamically selects a suitable access priority role, based on the dynamically calculated access rate of the application and the recent client's history (e.g. exceeding the time of calls beyond the agreed time threshold during the previous call, calling the server too often etc.). When the client application is calling the server for the first time, the server provides it with a fresh ID that must be presented in the following calls.

The WA design pattern package enables rapid prototyping of synchronization constraints. It is easy to customize the system by simply extending a list of policies that can be switched to at runtime.

7 Conclusions and Future Work

We have proposed a constraint language for separation of synchronization concerns from functional ones while keeping visible the role-oriented aspects of synchronization problems. Contrary to similar implementations, our solution allows the programmer to declare complex constraints, which can inspect (at runtime) the dynamic content of program variables and data.

We have demonstrated that it is possible to implement our approach in a general purpose programming language (OCaml) without using external tools. We can achieve this by analyzing and expressing common concurrency problems in the form of a design pattern; the patterns are then encoded in the host language as synchronization packages. We have implemented two example synchronization packages, where each package defines a given set of roles. One package has been used to implement a small application, which can dynamically switch between declared quality-of-service policies for accessing a web service.

We believe that our approach is language independent. However, the type system of OCaml certainly helped in representing policies in a way that resembles our formal specification of the constraint language in §3. In the future work, we would like to experiment with dynamically type-checked scripting languages like Python, or extensions of the OCaml language, such as Acute. Their dynamic binding features could make it possible to "plug-in" code of synchronization constraints at runtime. It may be also worthwhile to investigate the possibility of extending Java-like languages with the RBS approach, and using extensible compilers for the implementation of the constraint language.

We hope that our demonstration of declarative synchronization can be instructive, especially in the context of adaptive, non-stop systems, which may occasionally need to change their *modus operandi*.

References

1. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. In *Proc. ECOOP '02 (16th European Conference on Object-Oriented Programming)*, LNCS 2374, June 2002.
2. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, Oct. 1971.
3. F. L. Fessant and L. Maranget. Compiling join-patterns. In *Proc. HLCL '98 (Workshop on High-Level Concurrent Languages)*, 1998.
4. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. CONCUR '96 (7th Conference on Concurrency Theory)*, LNCS 1119, Aug. 1996.
5. S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proc. ECOOP '93 (7th European Conference on Object-Oriented Programming)*, LNCS 627, July 1993.
6. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.
7. M. A. Hiltunen and R. D. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, 1998.
8. W. Hursch and C. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Feb. 1995.
9. R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proc. ECOOP '03 (17th European Conference on Object-Oriented Programming)*, LNCS 2743, July 2003.

10. S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. POPL '96 (23rd ACM Symposium on Principles of Programming Languages)*, Jan. 1996.
11. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
12. C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Dec. 1997 (1998).
13. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
14. S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. Middleware 2003*, LNCS 2672, 2003.
15. G. Milicia and V. Sassone. Jeeg: A programming language for concurrent objects synchronization. In *Proc. ACM Java Grande/ISCOPE Conference*, Nov. 2002.
16. G. Milicia and V. Sassone. Jeeg: Temporal constraints for the synchronization of concurrent objects. Technical Report RS-03-6, BRICS, Feb. 2003.
17. Objective Caml. *http://caml.inria.fr*.
18. P. Panangaden and J. Reppy. The Essence of Concurrent ML. In F. Nielson, editor, *ML with Concurrency: Design, Analysis, Implementation, and Application*, pages 5–29. Springer, 1997.
19. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
20. Python. *http://www.python.org/*.
21. R. Ramirez and A. E. Santosa. Declarative concurrency in Java. In *Proc. HIPS 2000 (5th IPDPS Workshop on High-Level Parallel Programming Models and Supportive Environments)*, May 2000.
22. R. Ramirez, A. E. Santosa, and R. H. C. Yap. Concurrent programming made easy. In *Proc. ICECCS (6th IEEE International Conference on Engineering of Complex Computer Systems)*, Sept. 2000.
23. S. Ren and G. A. Agha. RTsynchronizer: Language support for real-time specifications in distributed systems. In *Proc. ACM Workshop on Languages, Compilers, & Tools for Real-Time Systems*, 1995.
24. P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Z. Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. Design rationale and language definition. Technical Report 605, University of Cambridge Computer Laboratory, Oct. 2004. Also published as INRIA RR-5329.
25. P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In *Internet Programming Languages*, LNCS 1686, pages 1–31, 1999.
26. B. Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994.
27. W3C. Web Services Architecture. *http://www.w3.org/TR/ws-arch/*.
28. P. T. Wojciechowski. Concurrency combinators for declarative synchronization. In *Proc. APLAS 2004 (2nd Asian Symposium on Programming Languages and Systems)*, volume 3302 of *LNCS*. Springer, Nov. 2004.
29. P. T. Wojciechowski and P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency. The Computer Society's Systems Magazine*, 8(2):42–52, April-June 2000.