A Component-Oriented Model for the Design of Safe Multi-threaded Applications

Reimer Behrends, R. E. K. Stirewalt and L. K. Dillon

Dept. of Computer Science and Engineering
Michigan State University

Abstract. We previously developed a component-oriented model that combines ideas from self-organizing architectures and from design by contract to address the complexity of design in multi-threaded systems. Components in our model are cohesive collections of objects that publish contracts declaring the conditions under which they access other components. These contracts localize a component's contextual synchronization dependencies in its interface. Moreover, the resulting systems permit strong guarantees of safety.

This paper reports a case study to validate the efficacy of our model on a realistic design problem: the component-based design of a multi-threaded web server. We first developed a bare-bones web server based on the Apache architecture and then subjected this design to three extension tasks. The study corroborates that our model enables a fine-grain component-based design of multi-threaded applications of realistic complexity, while guaranteeing freedom from certain synchronization errors.

1 Introduction

An essential property of a good component-based design is that component interfaces should make explicit all dependencies between components and the contexts in which they operate [1]. An important class of dependencies for systems in which multiple threads operate over shared objects relates to synchronization. Without thread synchronization, concurrent access to shared objects can lead to race conditions, and incorrect synchronization logic can lead to starvation and deadlock. However, synchronization policies and decisions are difficult to localize into a single software module—it is not uncommon for a module to implement a synchronization policy that satisfies safety and liveness requirements in some usage contexts but that fails to satisfy the same requirements in other contexts. Moreover, these contextual dependencies are difficult to record explicitly in a module's interface. These problems complicate the development of a component model with which to produce clean designs of concurrent systems.

We previously developed a solution that combines ideas from self-organizing architectures [2] and design by contract [3] to overcome these problems [4, 5]. Our *synchronization units model* associates with each module a declarative specification of its contextual synchronization dependencies, effectively localizing these dependencies in the module's interface. We designed these specifications as a non-obtrusive extension to existing object-oriented programming languages and component models, rather than as part of a stand-alone component model. Systems produced using this extension permits strong guarantees of safety. This paper presents a case study using this model.

In the synchronization units model, a component is a highly cohesive group of objects, called a *synchronization unit*, and components dynamically assemble and re-assemble into aggregates, called *realms*, that are associated with threads. For brevity, we refer to a synchronization unit as just a *unit*, and we henceforth use the terminology "unit" and "component" interchangeably. Each thread operates exclusively within its own dedicated realm. Inter-component dependencies represent access dependencies between the objects in a *client unit* and those of a *supplier unit*. Client units are *bound* to suppliers when the supplier *migrates* into the realm that hosts the client. Realms are self organizing; their dynamic assembly and evolution is governed by the negotiation of *synchronization contracts*, which a component designer declares as part of the client's interface. The self-organizing nature of this approach relieves the programmer from having to implement complex synchronization logic, and the explicit synchronization contracts declare all contextual dependencies that pertain to concurrency and synchronization.

To date, we have illustrated the theoretical power of the model on standard concurrency problems [4], integrated the model and its declarative contracts into an existing programming language [5], and shown how to safely integrate system libraries and third-party code (which may not be thread safe) with code written using contracts [6]. To validate the efficacy and applicability of our synchronization units model to current software engineering practice, we devised a realistically complex case study, which is the main focus of this paper. The case study involves a component-based design of a multi-threaded web server with database and scripting capabilities. The design is modeled after that of the popular Apache server.

Among the questions we expected to answer by the study are whether the synchronization units model is sufficiently expressive to handle designs of complex systems and how effective it is in localizing synchronization concerns. In an attempt to answer these questions, we subjected an initial design of a bare-bones web server to three maintenance and extension activities. These activities serve to validate that we can add complex new capabilities to the system by adding new units that require synchronization with existing units without modifying existing units. We describe two of these activities below. A description of the third can be found in [6] and is omitted here due to space constraints.

The rest of this paper is structured as follows. We first describe the web-server application that is the subject of our study (Section 2). We then introduce our model of synchronization units (Section 3) and outline the basic architecture of our component-based design of the web server (Section 4). We validate the effectiveness of our model on complex designs and the degree to which it localizes synchronization concerns by extending the bare-bones architecture with new features and assessing the ease with which our model supports the extensions (Sections 5–6). The paper concludes with a comparison of our approach to existing work (Section 7).

2 Rationale for choosing the application

The subject of this case study is a component-based design of the popular Apache web server. Apache itself incorporates a modular and extensible architecture in which software modules implement capabilities at many levels of granularity from http-request parsing to PHP scripting. While the Apache architecture was designed for extensibility, the afore-mentioned modules cannot be classified as software components because they fail to externalize all of their contextual dependencies. This deficiency is particularly noticeable as it regards issues of concurrency and synchronization. For example, the authors of the popular PHP scripting engine have warned against its use in the multi-threaded version of Apache [7], because the engine relies on a large number of third-party libraries that may not be thread-safe [8]. Other vulnerabilities have emerged in this multi-threaded version [9, 10]. Our model guarantees freedom from these sorts of synchronization vulnerabilities. This case study aims to demonstrate that our model can also accommodate the major design decisions that have made the Apache architecture so extensible—i.e., that Apache modules can be implemented as components in our model. Given the central importance of web servers in e-commerce applications, such a result could greatly increase the confidence in these applications.

Multi-threaded web servers comprise a rich set of interacting modules with unstructured synchronization dependencies. Many of these dependencies arise from security and efficiency concerns, which justify decisions to introduce shared resources. For example, a URI rewriting module may amortize the cost of translation by caching its results. Such caches must be protected from unsynchronized access by multiple threads. Large modules, e.g., internal scripting engines, are often shared because they are prohibitively expensive to replicate for each thread. Other dependencies arise from the use of non-reentrant third party libraries, such as the POSIX function `crypt`, which is used by an authentication module.

Because these modules contain implicit synchronization dependencies, it is difficult to certify the safety of a given configuration, a problem that is exacerbated by the reconfiguration requirements of modern web servers. Such applications are reconfigured often, e.g., to fix a bug, close a security hole, accommodate the changing requirements of evolving web standards, or service the varying needs of users [11, 12]. New modules may have synchronization needs and resource-access patterns that differ from the needs and patterns of the modules that they replace. Thus, assuring against concurrency flaws during maintenance and extension is a major problem, especially in cases where new modules are developed by third-party vendors. Such a problem begs for a component-based solution, but the component model must guarantee safety in the face of frequent reconfiguration. The remainder of this paper attempts to demonstrate how our model, which provides exactly these guarantees, can support the component-based design of such a large, extensible, and critical application.

3 Synchronization Contracts

This section overviews the key ideas underlying our model of synchronization units. Detailed discussions of the model can be found in [4, 5].

```
synchronization class REQUEST_HANDLER inherit PROCESS_BASE
feature { NONE } -- instance variables
  auth_stage, content_stage: BOOLEAN   -- condition variables
  authenticator: AUTHENTICATOR         -- unit reference
  connection: INBOUND_SOCKET           -- unit reference
  dispatcher: CONTENT_DISPATCHER       -- unit reference
  ...
feature { ANY } -- operations (methods)
  authenticate ( request : REQUEST ) is
  do
    auth_stage := true
    authenticator.validate(request)
    auth_stage := false
  end
  ...
concurrency -- concurrency clause follows
  connection
  auth_stage => authenticator
  content_stage => dispatcher
end -- class REQUEST_HANDLER
```

Fig. 1. Elided definition of the synchronization class REQUEST_HANDLER

In the synchronization units model, threads operate in disjoint realms that comprise one or more synchronization units. Any attempt by a thread to access a unit while that unit is not in its realm constitutes a *realm violation* and causes a run-time exception to be raised. The designer annotates a synchronization unit that plays a client role in a collaboration with one or more *concurrency constraints*, each of which specifies a condition under which the client requires exclusive access to a supplier. During execution, the run-time system assembles and reassembles realms, schedules threads to execute, and watches for realm violations using algorithms that 1) ensure that the realms are always pairwise disjoint, 2) guarantee that a thread executes only when the thread's realm consists of a *process root* unit and all suppliers required by constraints associated with units in the realm, and 3) avoids starvation and preventable deadlocks. Because realms are disjoint, two threads are assured to never concurrently access the same memory location—shared units migrate from one realm to another, but these units are never accessed simultaneously by different threads.

By way of illustration, we show part of the definition for a synchronization unit from the case study (Figure 1). It is written in an extension of Eiffel [13] with concurrency constraints, which we developed previously [4]. In this extension, the designer designates that certain classes are *synchronization classes* to indicate that they produce synchronization units when instantiated. In our web-server application, incoming web requests are handled and serviced by instances of a `REQUEST_HANDLER` class. The keyword `synchronization` (line 1) signifies that instances of this class define synchronization units.

The class declares several instance variables, some of which encode conditions under which a request handler requires exclusive access to a supplier (e.g., `auth_stage` and `content_stage` in line 3) and some of which reference the required supplier (e.g., `authenticator`, `connection`, and `dispatcher` in lines 4–6). We call the former *condition variables* and the latter *unit variables*. [1] For example, `auth_stage` records when a request handler needs to validate that a web request may access protected parts of a website; at such times, it requires the `AUTHENTICATOR` unit referenced by `authenticator` to validate that the access is permitted (line 12). A designer introduces condition variables to record the state of a unit's computation for use in concurrency constraints, thereby making the state observable by a run-time system that negotiates and enforces contracts.

Our Eiffel extension provides a *concurrency clause* for declaring contracts. The concurrency clause appears after the keyword `concurrency` (line 16). It contains three concurrency constraints (lines 17–19). Each concurrency constraint references a supplier unit, and the last two predicate the unit references on *guards*, which reference condition variables. Because it has no guard, the first constraint (line 17) declares an unconditional dependency—whenever a request handler executes, it requires exclusive access to the inbound socket referenced by `connection`. By contrast, the next two constraints (lines 18 and 19) say that the request handler accesses the unit referenced by `authenticator` (respectively `dispatcher`) only when the condition variable `auth_stage` (respectively `content_stage`) is true.

The run-time system for the extended language ensures that, when a process executes, its realm contains all the synchronization units that, according to the concurrency constraints, it needs to access. Briefly, the run-time system intercepts assignments to condition variables and unit variables and automatically reassembles so as to comprise only the required synchronization units. Should a process be unable to update its realm, because it requires exclusive access to a unit already in the realm of another process, it will be blocked until it can obtain the required exclusive access. For example, because the concurrency constraint in line 18 indicates that a process needs exclusive access to the unit referenced by `authenticator` when `auth_stage` is `true`, a process blocks if it attempts to execute line 11 (assign `auth_stage` the value `true`) when the indicated unit belongs to the realm of some other process. Later, when the indicated unit becomes available, the run-time system migrates the unit into the realm of the blocked process and unblocks the process. We describe algorithms to efficiently and fairly perform such *realm updates* in [5], along with the results of a performance study.

In controlling migration of units and scheduling of threads, the runtime system guarantees the invariance of the concurrency constraints in a program, thereby enforcing the contracts. This functionality is achieved without the programmer needing to write code that explicitly manipulates realm, unit, or thread representations. Instead, the program-

[1] In our Eiffel extension, a BOOLEAN variable used in a concurrency constraint is inferred to be a condition variable, and an attribute whose declared type is a synchronization class is inferred to reference a synchronization unit. For lack of space, we omit declarations of the synchronization classes `AUTHENTICATOR`, `INBOUND_SOCKET`, and `CONTENT_DISPATCHER`.

mer writes code that maintains condition variables to encode and reflect the current state of the computation and constraints that declaratively specify the suppliers that a unit accesses in each of these states. We contend that such code—by virtue of being local to a single unit—and concurrency constraints—by virtue of being declarative and expressed at appropriately high levels of abstraction—are less susceptible to synchronization errors than the typical code that a programmer writes to explicitly synchronize processes that share data. Moreover, errors due to code that does not conform to the stated contracts are easily detected at run-time by an efficient realm-boundary check. When one unit attempts to access another, the run-time system checks that the units belong to the same realm, allowing the access only if the check succeeds and raising a run-time exception otherwise.

4 Component-based design of a multi-threaded web server

Our architecture follows the basic design of the multi-threaded Apache web server, which manages a pool of threads to concurrently process HTTP requests as they arrive. Figure 2 depicts our architecture.[2] Borrowing ideas from [14–16], we use UML's built-in extension mechanisms to extend the UML class-diagram notation to express the structure of this architecture. Class names rendered in italics denote abstract classes. Our extensions involve three new stereotypes—⟨⟨synchronization⟩⟩, ⟨⟨process_root⟩⟩, and ⟨⟨external⟩⟩—that denote synchronization, process-root and external unit classes respectively. For brevity in this paper, our architectural drawings show only the synchronization classes; we therefore elide the ⟨⟨synchronization⟩⟩ stereotype in diagrams. Condition variables are shown as boolean-valued class attributes and unit variables as directed associations. We express concurrency constraints in curly braces adjacent to their associated synchronization class. For example, the `LISTENER_SOCKET` class declares the condition variable `startup`, the unit variable `namesvcs`, and the concurrency constraint `startup` => `namesvcs`.

The application comprises a main thread, called the *web dispatcher*, and a separate *request handler* thread for each incoming HTTP request. The realm of the web dispatcher is rooted by a unit of class `WEB_DISPATCHER` and includes a unit of class `LISTENER_SOCKET`, which it uses to monitor a port for connection requests. At times, specifically when the listener socket is "starting up," the realm of the web dispatcher also contains an external unit of class `NAME_SERVICES`, which is used to lookup IP addresses. The realm of each request handler is rooted by a unit of class `REQUEST_HANDLER`.

When a connection is requested, the web dispatcher creates an `INBOUND_SOCKET` unit with which to receive the actual HTTP request and communicate the resulting content. It then dispatches this unit to a new request handler thread. Each request handler thread parses the data sent over its assigned connection into a *request object*, handles the request, as described below, and then terminates. To minimize the overhead of request-

[2] Due to space limitations, Figure 2 abstracts away many details of the full case study—e.g. we omit the database capabilities. The actual web server developed for the case study consists of 57 Eiffel classes, of which 31 are synchronization classes, and a total of 3587 lines of code.

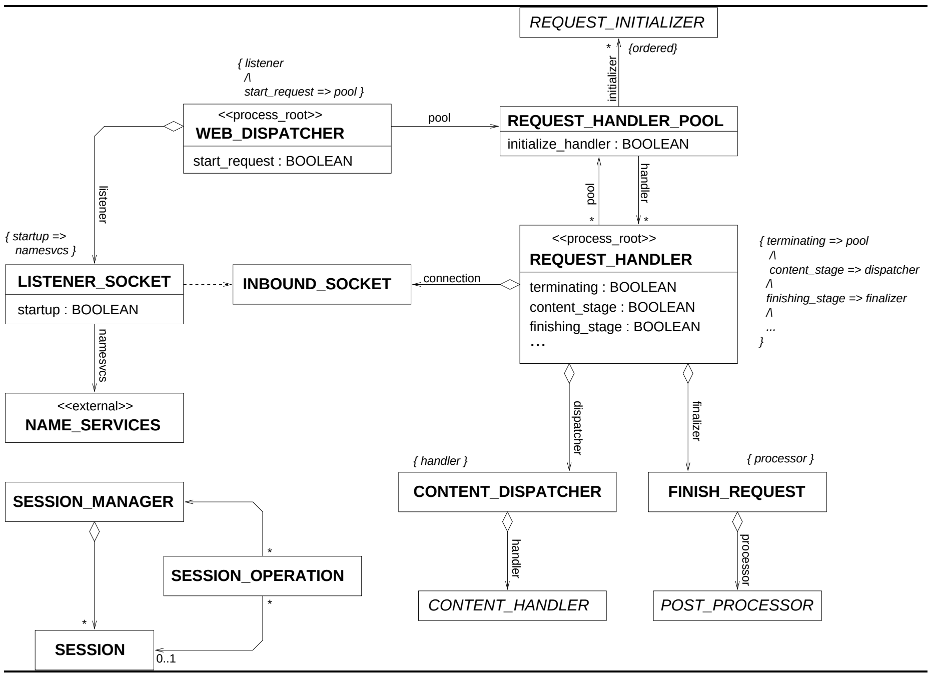

Fig. 2. Basic Web Server Architecture

handler initialization,[3] `REQUEST_HANDLER` units are stored in an object pool [17] and reused to process different HTTP requests. In our design, the web dispatcher actually dispatches inbound socket units to a unit of class `REQUEST_HANDLER_POOL`, which then selects or creates a new unit of class `REQUEST_HANDLER`, and creates a new thread rooted by that unit.

Request handling itself is implemented in stages. For brevity, Figure 2 depicts the components for only three of these stages—request initialization, content generation and request finalization—although our actual architecture supports others, such as authentication. *Request initialization* initializes a request object with attributes prior to parsing the actual request information that is arriving over the inbound socket. The abstract class `REQUEST_INITIALIZER` declares a method called `prepare` (not shown), which is parameterized by a reference to a request object. The bare-bones architecture uses a null request initializer, but many extensions need to initialize request objects with attributes before request handing begins in earnest.

Content generation generates the data—e.g., the contents of static web pages, the results of running a script, or an error report—to send back to the requester. This stage is performed by a `CONTENT_DISPATCHER` unit, which dispatches a request to one

[3] For example, each request handler must contain configuration information, such as which extensions are being used, which URIs need to be authenticated, etc. In our implementation, this information is retrieved from a central configuration file that is consulted when the system is initialized.

of several content handlers. Example handlers (not shown in diagram) include STATIC_HANDLER, which is used to serve static web pages, and ERROR_HANDLER, which is used for requests that resulted in an error. Other content handlers can be implemented as extensions to the abstract class CONTENT_HANDLER. The content generation stage uses the *builder* pattern [18], with CONTENT_DISPATCHER in the role of director and CONTENT_HANDLER in the role of the (abstract) builder.

Finalization is responsible for such things as logging and analysis. During this stage, the completed request object is passed to an instance of FINISH_REQUEST, which is responsible for the maintenance of log files, gathering of statistics, and general post-processing of completed requests. The FINISH_REQUEST unit also signals the REQUEST_HANDLER_POOL at the end of each request that a request handler is again available. Once the REQUEST_HANDLER unit is returned to the pool, the request handler thread is terminated. The finalization stage can be extended by adding components whose interfaces conform to the POST_PROCESSOR interface. In the bare-bones web server, only a simple logging mechanism is provided (not shown in the diagram).

The basic architecture also provides facilities for tracking and managing *sessions*, which are sequences of connected user-agent requests. The central SESSION_MANAGER unit controls access to a repository of SESSION units, each of which is identified by a unique session key. In this design, SESSION units comprise sets of key–value pairs that can be queried and set by components in the various stages of request handling. Often, client units do not use SESSION units directly, but instead access them through a SESSION_OPERATION facade, which simplifies access to session information by encapsulating the contract-negotiation and business logic required to locate and manage a session unit by key. SESSION_OPERATION components are constructed with a session key, which may be null. If the key is non-null, the component retrieves the corresponding SESSION from the SESSION_MANAGER; otherwise, it asks the SESSION_MANAGER to create a new session and returns the new session key. Subsequently, the SESSION_OPERATION facade forwards the querying and setting of key–data pairs to the retrieved SESSION. SESSION_OPERATION units encapsulate contract-negotiation logic that would otherwise need to be replicated among all SESSION clients, and they ensure minimal contention for the singleton SESSION_MANAGER unit.

As presented thus far, our basic architecture employs a component-based design in order to support extension and to simplify static reconfiguration. However, the loose coupling and fine-grain decomposition of functionality that make the design so extensible is at odds with attempts to make components thread safe and deadlock free. We hypothesize that our synchronization units model enables a fine-grain component-based design (and the concomitant benefits with regard to maintenance and extension/contraction) while guaranteeing freedom from data races [19] and automatically avoiding or recovering from most classes of deadlock. To test this hypothesis, we subjected the basic architecture to three different maintenance tasks, each involving a non-trivial extension comprising one or more component collaborations that interact with existing collaborations in the basic architecture. We describe two of these activities below. For each extension, we document the business logic requirements, provide a design of the extension in our model, and summarize the capabilities demonstrated by the extension.

5 Extension: Dynamic Content

Our first maintenance task extends the web server with a scripting facility for generating dynamic content. Scripts must be able to safely access web-server resources, especially session data. Moreover, scripts typically access such resources according to a two-phase locking protocol, whereby all shared resources are acquired before any are released [20]. Such resources manifest as synchronization units in our basic architecture. Thus, this maintenance task aims to see if we can implement scripting as a set of new components that collaborate and safely synchronize with the existing components without having to modify any of those existing components. A related issue concerns the need for scripts to access standard library functions, such as the POSIX function `crypt` and DNS functions, many of which are not thread safe. In prior work, we showed how multi-threaded accesses to standard libraries can be serialized using wrapper facades that coordinate with external synchronization units [6]. This maintenance task builds upon these prior results.

5.1 Scripting language and its embedding

To explore these issues, we chose to support scripts written in the Lua language [21]. Lua is a small interpreted language with metaprogramming facilities for extending the language with new features and hooks into a host application, such as a web server. We chose Lua over languages such as PHP for purely pragmatic reasons: The Lua integration shares in all of the the essential complexity that would occur in a PHP integration with far less accidental complexity.

Using its metaprogramming facilities, we extended Lua with new primitives for 1) accessing standard libraries and resources in our web-server architecture, and 2) declaring resource needs, as dictated by the two-phase locking protocol. Briefly, we represent each library and each web-server resource as a Lua object, hereafter called a *resource proxy* that is visible to user scripts. For example, we provide a Lua object named mySession, which represents a `SESSION` unit in the basic architecture. When a Lua script is executed in response to some http request, mySession is bound to the `SESSION` component associated with the request. Among others, we also provide the Lua objects myCrypt and myNameservices, which represent `CRYPT` and `NAME_SERVICES` units.[4]

In addition to these resource-proxy objects, we extended the Lua language with a new statement called *acquire*, which a script programmer invokes to declare the resources he or she intends to access. The statement takes a variable number of arguments, all of which must be resource proxies, such as mySession or myCrypt. Semantically, we interpret an `acquire` statement as a request to atomically acquire the named resources; a running script blocks on such a statement until all of the named resources

[4] Note that these components are external synchronization units that serialize accesses to the POSIX function crypt and functions in the DNS library (See [6] for details).

```
(1)  acquire(mySession, myCrypt)
(2)  if not mySession.get("auth_flag")
(3)  then
(4)    if myCrypt.crypt(...) == ...
(5)    then
(6)      ...
(7)    end
(8)  end
(9)  ...
```

(a) Example Lua script

```
feature acquire_units(
  use_crypt,
  use_nameservices,
  use_session: BOOLEAN) is
do
  acquire_crypt,
  acquire_nameservices,
  acquire_session :=
    use_crypt,
    use_nameservices,
    use_session
end
```

(b) `LUA_INTERPRETER` support function

Fig. 3.

have been acquired. Figure 3(a) depicts a small example. Line (1) states the script programmer's intention to access information about the current session and to invoke the crypt function. The session information is queried on line (2) and the crypt function is invoked on line (4). All resources acquired by a script are released automatically when the script completes. Thus all Lua scripts conform to the two-phase locking protocol.

5.2 Component-based solution

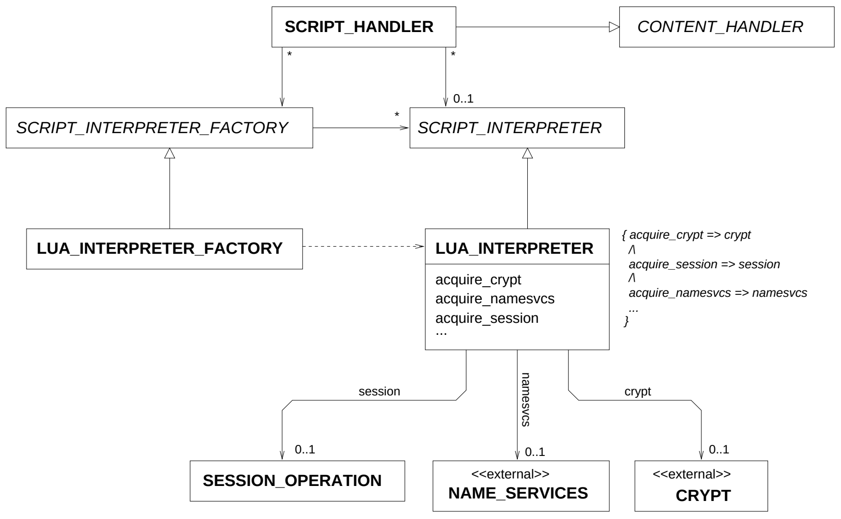

Fig. 4. Dynamic Scripting Extension

Figure 4 depicts our extension. For brevity, this diagram depicts the unit classes and associations that are new to this extension and only those classes from the basic architecture upon which these new classes depend. Scripting is supported by a new content handler called `SCRIPT_HANDLER`, which uses a `SCRIPT_INTERPRETER` component to actually execute a user script. `SCRIPT_INTERPRETER` is an interface class. Concrete components that conform to this interface implement interpreters for specifi c scripting languages; the `LUA_INTERPRETER` component is an example. To support extension, we designed `SCRIPT_HANDLER` to bind to a particular `SCRIPT_INTERPRETER` component according to the *abstract factory pattern* [18]. `SCRIPT_INTERPRETER_FACTORY` plays the abstract-factory role in this design. This abstract class provides a create method (not shown) that is parameterized by a `REQUEST` object. Concrete factories, such as `LUA_INTERPRETER_FACTORY`, may then access request-specifi c information when deciding which interpreter to create or retrieve from a repository.

Class `LUA_INTERPRETER` has two salient characteristics. First, it is a wrapper-facade [22] that encapsulates the data structures (i.e., C-style structs) that implement a Lua interpreter, and it provides an object-oriented interface to the ANSI C functions that operate over these structures. Second, it is a synchronization class that declares condition variables and parameterized contracts with other components in the architecture. These condition variables are manipulated when the interpreter executes an acquire statement in a Lua script, thereby setting the condition variables that parameterize the relevant contract.

Operations in a Lua script that manipulate resource proxies are connected to operations over synchronization units as follows. `LUA_INTERPRETER` units link to the units that correspond to the resource proxies. Suppose, for brevity, that there are only the three proxies mySession, myCrypt, and myNameservices. Then class `LUA_INTERPRETER` associates to a `SESSION_OPERATION` unit, a `CRYPT` unit and a `NAME_SERVICES` unit, as depicted in Figure 4. Moreover, for each such association, x, class `LUA_INTERPRETER` declares a condition variable `acquire_x` and the contract:

```
acquire_x => x
```

Thus, `LUA_INTERPRETER` units are able to reflect the resource acquisition needs of Lua programmers, and changes in these acquisition needs trigger a renegotiation of contracts with the corresponding units.

`LUA_INTERPRETER` implements `acquire` by atomically setting the appropriate condition variables via a function such as that depicted in Figure 3(b). In our extended version of Eiffel, several variables can be assigned new values atomically, and synchronization contracts are renegotiated only after the entire assignment completes. If these contracts cannot be negotiated, the thread blocks on this statement, thereby providing the expected semantics of the Lua `acquire` statement. Consider, for example, the call to `acquire` in line 1 of Figure 3(a). This call produces a call to `acquire_units` with the value `true` for the fi rst and third parameters and the value `false` for the second. Thus, the assignment in `acquire_units` blocks until the `SESSION_OPERATION` and `CRYPT` units are migrated into the realm hosting the interpreter.

5.3 Discussion

This task demonstrates that our model supports the safe addition of complex functionality to an existing design without having to modify existing components. We added scripting capabilities to the bare-bones web server without modifying any previously existing components. It is unlikely that we could have added the same capability to the multi-threaded Apache architecture without modifying any existing modules. At a minimum, we would have to carefully analyze the augmented system for potential race conditions and deadlocks, which would involve a detailed analysis of the code for these modules. By contrast, our model guarantees freedom from data races. In ongoing research, we are also developing techniques for analyzing synchronization dependencies between units, as expressed in their concurrency clauses, to expose unpreventable deadlocks or show that no unpreventable deadlocks can occur.

In addition, our implementation allows script writers to reap many of the benefits of the underlying synchronization units model. For example, whereas user scripts may use resources in patterns that we could not anticipate when designing the scripting components, the resulting system is still guaranteed to be free of data races. If a script author forgets to acquire a resource before attempting to use it, the interpreter raises a run-time exception rather than permitting the access.

6 Extension: Load Balancing

Our second maintenance task extended the dynamic content handler with a load balancing mechanism. In this extension, the total number of available script interpreters is capped and adjusted at run time as a function of the server load and static configuration parameters. The extension aims to demonstrate two aspects of the web server design obtained using synchronization contracts: 1) it can accommodate quality-of-service improvements without incurring massive re-design, and 2) it admits replacement of one component with a new one, although the new component and the replaced component have different contextual synchronization dependencies, without requiring changes to other components.

6.1 Number of scripting interpreters

The load balancing mechanism automatically estimates the number of interpreters needed according to a heuristic that makes use of a configuration constant M, which is set by the server administrator, and two computed values, the approximate number R of requests for dynamic content per second and the approximate average time T to process a request for dynamic content. The heuristic aims to ensure that, on average, M interpreters are available for each thread. Threads are assigned idle interpreters whenever possible and are randomly assigned a busy interpreter otherwise.

6.2 Component-based solution

Figure 5 depicts the new units and collaborations in our extension. The new unit `LOAD_BALANCER` is a singleton `POST_PROCESSOR` component that processes each request dur-

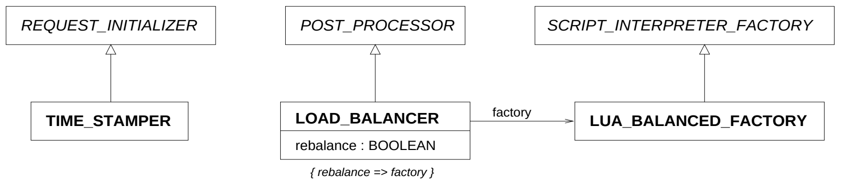

Fig. 5. Load Balancing

ing the finalization stage of request handling. This new unit computes the approximations R and T and, at regular intervals, initiates a *rebalancing process*, during which it computes a new target number of interpreters and informs the factory (which is used by SCRIPT_HANDLER) of this change. This extension replaces the LUA_INTERPRETER_FACTORY unit in Figure 4 with a new unit called LUA_BALANCED_FACTORY, which contains extra functionality to collaborate with the load-balancing components. During rebalancing, LOAD_BALANCER notifies the LUA_BALANCED_FACTORY unit of changes in the target number of LUA_INTERPRETER units in the repository. In response, LUA_BALANCED_FACTORY creates new interpreter components or deletes superfluous components to achieve the target number.

This extension requires a REQUEST_INITIALIZER called TIME_STAMPER, which augments a REQUEST object with a time stamp that the LOAD_BALANCER consults when finalizing the request object to derive an estimate for T. This new extension involves one new synchronization contract, which specifies that when rebalancing is in progress, LOAD_BALANCER requires exclusive access to LUA_BALANCED_FACTORY.

6.3 Discussion

This task demonstrates two useful aspects of our synchronization units model. First, we showed how a component that participates in one collaboration can be replaced with another component that participates in an additional collaboration without having to modify the synchronization logic in any of the client components in either collaboration. In this case, we swapped out the LUA_INTERPRETER_FACTORY unit for the new LUA_BALANCED_FACTORY unit. This exchange was trivial—we did not alter the factory's client (i.e., SCRIPT_HANDLER) or its interface (i.e., the abstract class SCRIPT_INTERPRETER_FACTORY). This exchange was so trivial because components in our model explicitly represent synchronization dependencies in their interfaces.

Second, and perhaps more interesting, this task demonstrates the improvement of quality of service of an existing design without having to fundamentally alter the design or modify many of its existing components. In fact, the only modification was the component replacement already mentioned. Of course, this is only one example, but it is interesting that a post-hoc quality-of-service optimization was so easy to incorporate into a design that was built upon a component model with support for synchronization. We are currently investigating whether other QoS optimizations are simplified by virtue of our model's ability to deal with synchronization concerns.

7 Conclusions and Future Work

This case study demonstrates our synchronization units model supports the component-based design of a realistically complex multi-threaded system. Our maintenance tasks involved the extension of a component-based design with new features that imposed new synchronization requirements on that existing design. In both cases, the extensions were accommodated by reusing and replacing—but never modifying or redesigning—the existing components in the bare-bones architecture. By virtue of the guarantees inherent to the synchronization units model, the resulting variants are free of data races. Moreover, by virtue of the algorithms used to implement the dynamic assembly and re-assembly of realms, the resulting variants automatically avoid and/or recover from preventable deadlocks. We now discuss the issues that we believe contributed to the ease with which these maintenance tasks were accomplished.

One key question for concurrency management in component-based approaches is how to associate synchronization logic with components. Briefly, synchronization logic can be either associated with the component that is a supplier of services or with the component that is the client of that supplier. Traditionally, it has been held that encapsulating synchronization logic in the supplier leads to the most modular and extensible architecture [23, 24]. Moreover, many of the existing client-side approaches are vulnerable to race conditions [25, 26].

Our model goes against the conventional wisdom by intentionally attaching synchronization logic to the component that plays the role of the client in a client–supplier relationship. Indeed, we believe that this approach enables the development of more extensible component architectures. There are two reasons for this. First, for any component of reasonable size, the component developer may not be able to anticipate all of the patterns of service invocations by clients. The knowledge of how the services are being used—in particular, which sequences of service invocations on a component have to be treated as an atomic operation—resides within the clients that use these services. Second, a client may call upon more than one supplier component in the course of an atomic operation, such as how `LUA_INTERPRETER` uses a number of formerly unrelated components of our basic server architecture. Naturally, this information is not easily encapsulated in a single supplier component (though Holmes [24] introduces the concept of synchronization rings—wrappers around sets of one or more suppliers—to make this possible). As we have seen in our case study, extensibility seems to benefit rather than suffer from client-side synchronization. Instead of requiring us to add post-hoc extensions to existing components, client-side synchronization allowed us to reuse existing components in formerly unanticipated ways.

The cost of attaching synchronization information to the client component is that synchronization logic may have to be unnecessarily replicated for each client. Our approach combats this risk in two ways. First, we made our contract language compact and declarative; so in most cases there is little or no replication to begin with. Second, we allow the reuse of synchronization logic through the usual techniques, such as aggregation and class extension. Consider, for example, our session-management facilities, in which clients must retrieve `SESSION` components from a centralized `SESSION_MANAGER`. If k distinct client components were each to interact directly with `SESSION`s and `SESSION_MANAGER`s, then yes, the synchronization logic would need to be replicated in k different synchronization classes. However, when this occurs, the synchronization logic can be localized and encapsulated within an additional component, such as `SESSION_OPERATION`. By declaring a contract with these special components, clients may reuse rather than replicate these complex patterns of synchronization logic.

Our future work will focus on stress-testing the extensibility of component architectures designed with our model. Specifi cally, we will experiment with more sophisticated quality-of-service optimizations that are diffi cult to localize in a single component or cohesive group of components. For example, the load balancing mechanism presented in Section 6 introduces a potential bottleneck in the request processing pipeline, because each request handler must access the interpreter factory twice. Eliminating this bottleneck means shifting part of the work towards the beginning or the end of the pipeline (where contention already exists and is unavoidable). Such an extension requires adding a set of components at normally unrelated stages of the pipeline.

We are also looking at integrating our contractual approach with existing component models, such as the Corba Component Model (CCM) [27] or Enterprise Java Beans (EJB) [28], with an eye towards automated handling and reasoning of concurrency properties of large component assemblies. Reasoning about such assemblies manually can be overwhelming, if not infeasible [29]. However, synchronization contracts already automate the handling of some synchronization properties, and by virtue of providing a (partial) specifi cation of the concurrent behavior of components, could be leveraged to reason automatically about even more diffi cult non-local properties, such as liveness issues. To this end, we have conducted some preliminary experiments on fi nding deadlocks at compile time by analyzing the synchronization contracts of components, with the goal of augmenting our runtime deadlock-avoidance and recovery algorithms. Similarly, synchronization contracts could be used to assist EJB programming by replacing the automated concurrency handling through containers with a contractual approach. Existing research indicates that the use of containers for concurrency control can be a serious performance bottleneck for using entity beans in EJB applications [30]. Providing an explicit concurrency control mechanism with strong guarantees for the avoidance of race conditions and deadlocks such as ours might be able to resolve such bottlenecks.

Acknowledgements. Partial support for this research was provided by the Office of Naval Research grant N00014-01-1-0744 and by NSF grants EIA-0000433 and CCR-9984726.

References

1. Szyperski, C.: Component software: Beyond object-oriented programming. Addison–Wesley (2002)
2. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: Proc. of the 4^{th} ACM SIGSOFT Symposium on the Foundations of Software Engineering. (1996)
3. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1997)
4. Behrends, R., Stirewalt, R.E.K.: The universe model: An approach for improving the modularity and reliability of concurrent programs. In: Proc. of ACM SIGSOFT Symp. on the Foundations of Software Engineering (FSE-00). (2000) 20–29
5. Behrends, R.: Designing and Implementing a Model of Synchronization Contracts in Object-Oriented Languages. PhD thesis, Michigan State University (2003)

6. Behrends, R., Stirewalt, R.E.K., Dillon, L.K.: Avoiding serialization vulnerabilities through the use of synchronization contracts. In: Proc. of the Workshop on Specification and Automated Processing of Security Requirements. (2004) Held in conjunction with the IEEE Intl. Conf. on Automated Software Engineering.
7. Lerdorf, R.: PHP and Apache2 (2004) http://news.php.net/php.internals/10491.
8. The Apache Software Foundation: (Apache 2.0 thread safety issues) URL:http://httpd.apache.org/docs-2.0/developer/thread_safety.html.
9. Common Vulnerabilities and Exposures (CVE) Editorial Board : Candidate number 2003-0189 (2003) URL:http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0189.
10. Common Vulnerabilities and Exposures (CVE) Editorial Board : Candidate number 2003-0789 (2003) URL:http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0789.
11. Ricca, F., Tonella, P.: Analysis and testing of web applications. In: Proceedings of the 23rd International Conference on Software Engineering, IEEE (2001)
12. Sabbah, D.: Software engineering and the internet. In: Proceedings of the 23rd International Conference on Software Engineering, IEEE (2001) Keynote speech.
13. Meyer, B.: Eiffel: the Language. Prentice Hall (1992)
14. Abi-Antoun, M., Medvidovic, N.: Enabling the refinement of a software architecture into a design. In: Proc. of 2^{nd} Intl. Conf. on the Unified Modeling Language (UML). (1999)
15. Kaveh, N., Emmerich, W.: Deadlock detection in distributed object systems. In: Proc. of ESEC/FSE 2001. (2001)
16. Egyed, A., Medvidovic, N.: Consistent architectural refinement and evolution using the unified modeling language. In: Proc. of the 1^{st} Workshop on Describing Software Architecture with UML. (2001)
17. Grand, M.: Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML. John Wiley and Sons (1998)
18. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison–Wesley, Reading, Massachusetts (1995)
19. Netzer, R.H.B., Miller, B.P.: What are race conditions?: Some issues and formalizations. ACM Letters on Programming Languages and Systems **1** (1992) 74–88
20. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Communications of the ACM **19** (1976) 624–633
21. Ierusalimschy, R.: Programming in Lua. Lua.org (2004)
22. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture Volume 2 – Networked and Concurrent Objects. John Wiley and Sons (2000)
23. Bloom, T.: Evaluating synchronisation mechanisms. In: Seventh International Symposium on Operating System Principles. (1979) 24–32
24. Holmes, D.: Synchronisation Rings - Composable Synchronisation for Object-Oriented Systems. PhD thesis, Macquarie University, Sydney (1999)
25. Hoare, C.A.R.: Communicating Sequential Processes. Prentice/Hall International, Englewood Cliffs, New Jersey (1985)
26. Hansen, P.B.: Java's insecure parallelism. ACM SIGPLAN Notices **34** (1999)
27. Object Management Group: Corba component model, v3.0 (2002) http://www.omg.org/technology/documents/formal/components.htm.
28. DeMichiel, L., Yalcinalp, L.U., Krishnan, S.: The Enterprise JavaBeans 2.0 specification (2001) http://java.sun.com/products/ejb/docs.html.
29. Ranganath, V.P., et al.: Cadena: enabling CCM-based application development in Eclipse. In: OOPSLA Workshop on Eclipse Technology eXchange. (2003) 20–24
30. Cecchet, E., Marguerite, J., Zwaenepoel, W.: Performance and scalability of EJB applications. In: Proc. of ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. (2002) 246–261