# The Universe Model: An Approach for Improving the Modularity and Reliability of Concurrent Programs

Reimer Behrends
Dept. of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824
+1517 355 1246
behrends@cse.msu.edu

R. E. Kurt Stirewalt[1]
Dept. of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824
+1517 355 2359
stire@cse.msu.edu

## ABSTRACT

We present the universe model, a new approach to concurrency management that isolates concurrency concerns and represents them in the modular interface of a component. This approach improves program comprehension, module composition, and reliability for concurrent systems. The model is founded on designer-specified invariant properties, which declare a component's dependencies on other concurrent components. Process scheduling is then automatically derived from these invariants. We illustrate the advantages of this approach by applying it to a real-world example.

## Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; D.2.4 [**Software Engineering**]: Program Verification—*assertion checkers, reliability*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures.*

## General Terms

Design, Languages, Reliability

## Keywords

Distributed and Parallel Systems, Reliability, Software Architecture, Component-based Software Engineering

## 1. INTRODUCTION

Parallel and distributed systems are becoming increasingly important. Unfortunately, the increased expressive power afforded by the use of concurrency comes at the cost of increased complexity. Whereas many sequential programming languages now provide mechanisms for detecting common programming errors, such as dangling pointers and indexing problems, analogous mechanisms for frequent errors in concurrent systems—such as absence of mutual exclusion or temporal ordering problems—are often not available, not scalable, or too difficult for the average programmer to use correctly. In this paper, we present a novel concurrency-management mechanism that is easy and simple to use, that scales well even for multi-layered applications with a complex structure, and that provides a strong guarantee of reporting errors that exist implicitly or explicitly in either the code or the specification of concurrent problems.

Our approach, called the *universe model*, addresses a fundamental deficiency in many existing concurrent systems, namely that concurrency management tends to be hidden deep inside procedural code. While certain synchronization aspects may be exposed as part of the modular interface (such as declaring that a procedure has monitor semantics), safety constraints on the use of sequences of such services often either cannot be expressed in the interface or result in an interface description that exposes too many implementation details. As a result, the implementation of concurrency management is heavily delocalized. Modules do not compose cleanly or symmetrically, and their usage tends to rely on ad-hoc, resource-oriented techniques, making it hard to analyze them for errors.

The basic idea of our alternative approach is to describe the interactions and resource requirements of an application on a per-class basis using a simple specification language based on propositional logic, and similar in concept to the axioms of an abstract data type. The system then automatically derives scheduling information and access-protection logic from this specification. By inferring concurrent behavior from correctness conditions, mismatches between specified and actual behavior can be automatically recognized and reported.

Our declarative approach aims to achieve three separate objectives. First, simplify the task of program comprehension for concurrent programs by separating and abstracting concurrency concerns. Second, enable modular reasoning and composition in the presence of concurrency. Third, use the redundancy of specifying concurrency explicitly in addition to the implicit behavior present in the code to detect errors more reliably.

Our approach is grounded on two core principles: partitioning a program's data space into a set of disjoint subspaces, and requiring sequential execution within any such subspace. While these principles are similar in concept to the characteristics of a message-passing

[1] Partial Support provided by NSF grant CCR-9984727.

system, we assume procedure call semantics and, more importantly, let our specification language define how these subspaces interact.

## 2. CONCURRENCY MANAGEMENT

The common theme among all concurrency-management approaches is to *sequentialize* parts of the program. Monitors, for instance, allow only one process to execute within the monitor; message passing systems assume a number of interoperating sequential processes, which block on synchronizing statements until all participants are ready. Unfortunately, in all these approaches, complexity increases dramatically as the size of a program and the amount of concurrency scale up. This section discusses three sources of complexity, which our model addresses. First, existing mechanisms provide primitives that make it difficult to separate the management of concurrency from the underlying computation. Consequently, concurrent programs are often difficult to understand (2.1). Second, in addition to complicating program comprehension, the interleaving of these primitives into computation code has deleterious effects on the reusability of modules whose implementations use concurrency (Section 2.2). Finally, this problem is particularly acute in modules that provide services requiring multiple steps (Section 2.3).

### 2.1 Interleaving Concurrency and Computation

*Interleaving* expresses the merging of two or more distinct program plans within some contiguous textual area of a program [18, 17]. In this definition, a program plan refers to a description or representation of a computational structure that has been proposed to achieve some purpose or goal in a program [12, 16]. We view the management of concurrency as one of many plans that contribute to the design of a concurrent system. Viewed in this light, existing approaches to concurrency management do not support the direct exposition of such plans. Consequently, the *concurrency management plan* tends to be interleaved with other computational plans in the system.

To illustrate this, we begin by discussing a classical concurrent problem: the Dining Philosophers. Textbook solutions [20, 3] usually involve tricky resource manipulations to ensure the atomic acquisition of both forks while avoiding starvation, whereas the basic concurrency plan is straightforward: Whenever a philosopher eats, he requires access to both the fork on his left-hand side and the fork on his right-hand side. In a semi-formal notation, we might express this plan as follows:

$$\textit{eating} \Rightarrow \textit{left\_fork} \wedge \textit{right\_fork}$$

Figure 1 (from [3]) illustrates a monitor-based solution[1] of the dining philosophers problem [20]. Observe that our well-intentioned concurrency plan has been interleaved with the underlying computation in a way that severely reduces the readability of this code.

```
monitor Fork_Monitor
  Fork: array(0..4) of Integer range 0..2:=(others=>2);
  OK_to_eat: array(0..4) of Condition;

  procedure Take_Fork(I: Integer) is
  begin
    if (Fork(i) /= 2 then Wait(OK_to_Eat(I)); end if;
    Fork((I+1) mod 5) := Fork((I+1) mod 5)-1;
    Fork((I-1) mod 5) := Fork((I-1) mod 5)-1;
  end Take_Fork;

  procedure Release_Fork(I:Integer) is
  begin
    Fork(I+1) mod 5) := Fork((I+1) mod 5)+1;
    Fork(I-1) mod 5) := Fork((I-1) mod 5)+1;
    if Fork((I+1) mod 5)=2 then
      Signal(Ok_to_Eat((I+1) mod 5);
    end if;
    if Fork((I-1) mod 5)=2 then
      Signal(Ok_to_Eat((I-1) mod 5);
    end if;
  end Release_Fork;
end Fork_Monitor;
```

**Figure 1: Dining Philosophers using Monitors**

The operations that implement picking up and releasing a fork are placed inside a monitor, which guarantees that only one process at a time can execute the code. Unfortunately, because the concurrency plan requires two objects to be acquired, we must interleave knowledge of the object topology—namely, the circular organization of philosophers and forks—in order to correctly implement the plan. Such an interleaving fails to capture the essence of the problem. Moreover, the concurrency management details have been merged with the program code in a way that makes it close to impossible to identify concurrent interaction. It would be very difficult, for example, for someone unfamiliar with the problem to derive the simple declarative specification of the concurrency plan by reading the code. Instead of abstracting and separating this information, the monitor-based solution delocalizes it.

A side effect of this interleaving is a loss of reliability. Code such as that in Figure 1 is hard to verify, and the effect of changes is not always apparent. The solution in Figure 1 has no mechanism to indicate whether or not the program achieves proper mutual exclusion. In fact, moving the procedures to take forks and to release forks outside this monitor may appear to work just as well. The problem will go unnoticed until a race condition occurs.

### 2.2 Effect on Modularity

The interleaving of concurrency and computation plans also has a deleterious effect on the *composability* of modules whose implementations use concurrency primitives. We cannot just plug modules together and assume that concurrent interaction will work correctly. This composition problem is particularly acute in hierarchical systems, where the need to specify process interaction can easily compromise the data abstraction of a module's interface. For instance, suppose two processes $P1$ and $P2$ (Figure 2) access a common database $D$ through separate frontends $A$ and $B$. To make the database visible in the interfaces of $A$ and $B$ would violate the principle of information hiding: Their interface to module $D$ should not be visible to the implementations of $P1$ and $P2$. But if the database isn't made visible, non-atomic transactions by $P1$ and $P2$ could interfere with one another.

[1]This should not obscure the fact that more sophisticated solutions exist [1]; however, we believe that the problems discussed here are symptomatic of many approaches that—in whatever guise—rely on critical sections of code. In particular, it appears to be hard to separate concurrency control from computation in imperative languages without causing interleaving or delocalization problems.

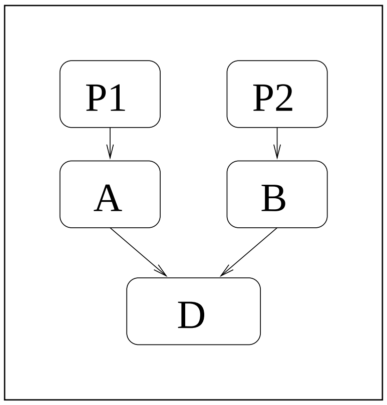


**Figure 2: A Modularity Problem**

## 2.3 Multi-Step Transactions

A *multi-step transaction* is a sequence of service invocations by and among multiple objects to perform some task. We use the word transaction to suggest that the sequence should either run to completion or not at all; that is, we want to remove from concern any sequence that is interrupted or interleaved by service requests from other concurrent processes.

Many languages guard transactions with a mechanism for *synchronizing* access to an object for the duration of such a transaction. To understand how this works, consider the following fragment of pseudocode:

```
synchronized ob do
  if ob.query then
    ob.action
  end
end
```

The `synchronized` statement directs the run-time system to execute the entire block of statements atomically with respect to `ob`. Consequently, the process may perform a multi-step transaction without concern for interleaved invocations from other (concurrent) processes.

Unfortunately, the proper use of such a feature cannot be checked by a compiler and so requires designer discipline to use correctly. Consider, for example, that the programmer forgot to encapsulate the block of statements inside the **`synchronized`** `ob` **`do`** … **`end`** construct. The program would compile without error, and the omission might go undetected for many runs until some unfortunate timing sequence leads to an undesired scheduling of processes. A compiler cannot determine the need for `synchronized` blocks because the modular interface to `ob` does not contain information from which transaction sequences (i.e., sequential necessity) can be derived. With a traditional modular interface, the specification of transaction necessity is hidden inside the procedural implementation, rather than being part of a publically available interface. Moreover, the use of features like `synchronized` violates the principle of information hiding: Information about the implementation of `ob` must be exposed in order for clients to use it correctly.

Features such as `synchronized` are symptomatic of the generally poor support that modern mainstream programming languages provide for information hiding in the presence of concurrency. The module interface of `ob` lists a set of services with no hint of constraints on the sequence of invocation of these services. Because interfaces usually do not represent sequencing constraints, there is no way for a compiler to detect the absence of a synchronized statement or to enforce the proper sequencing at run time. However, to avoid an error, a programmer must know to guard a sequence of `ob`-service invocations, such as that shown in the statement block, from interleaved `ob` calls by other concurrent processes.

## 3. UNIVERSE MODEL

Our *universe model* addresses each of the problems raised in section 2. In this model, concurrent processes run in isolation and do not engage in inter-process synchronization. Instead, each process is dynamically assembled from single-threaded object sets, called *universes.* In lieu of inter-process synchronization, an underlying run-time scheduler migrates universes among different processes according to designer constraints. This process of dynamic assembly allows the creation of composable concurrent systems.

### 3.1 Objects, Universes, Processes, and Realms

A *universe* is a set of objects. Every object in the system belongs to exactly one universe. Conceptually, universes partition the global data space into a set of local data spaces. Each universe *hosts* a process, and no universe may host more than one process at a time. A process that is hosted by a universe is said to *own* the universe[1]. In our model, it is illegal for code that is running in a process $P$ to access an object in a universe that is not owned by $P$. Such an access results in a run-time exception.

The *realm* of a process is the set of universes that the process owns. Realms may grow or shrink during the lifetime of the process. The allocation of universes to processes—that is, the manipulation of process realms—is managed by an underlying run-time scheduler, which migrates universes among processes in order to satisfy *realm constraints.* Briefly, realm constraints implicitly specify the set of universes that a process must own in order to continue execution. If the realm constraint of process $P$ includes a universe that is part of another process' realm, then $P$ blocks until the universe becomes available. Deadlock occurs if there is a cycle of processes, each blocked and waiting for the next process in the cycle to release a universe. Realm constraints are computed dynamically from specifications that are defined by the designer for each universe (See Section 3.2).

It is important to note that there is a difference between a process accessing a universe (say, through a function call), and owning it. For a process $P$ to legally access a universe $u$, $P$ must own $u$. If $P$ does not own $u$, but code in $P$ tries to access $u$, then a runtime exception will be raised. On the other hand, if $P$ does not own $u$, but the realm constraint of $P$ includes $u$, then $P$ will block until such time as $u$ can be added to the realm of $P$. Observe that acquiring a universe never results in a runtime error (as long as it doesn't cause deadlock).

[1]For the purpose of our definition, a process is a thread of control operating on some data. We assume in particular that a process has no sub-processes.

These conditions guarantee that the data spaces of any two processes cannot overlap. Consequently, we think of programs in the universe model as a set of mostly independent sequential processes.

## 3.2 Concurrency Constraints

Realm constraints, which are used by the run-time scheduler to migrate universes among different processes, are computed dynamically from simpler constraints called *universe invariants.* Each universe is accompanied by an invariant specification, which is a propositional formula that specifies the universe's access dependencies on other universes. Atomic propositions in these formulae can be one of the following:

– A boolean variable (also called a condition variable).

– A universe variable.

Condition variables are used by programmers to encode certain observable states of a computation. Designers should identify condition variables that precisely describe the conditions under which the universe requires access to other universes.

In the dining philosophers example, we identified a condition variable called *eating*, which when true implies an access dependency on two other universes (*left_fork* and *right_fork*). We specify such an invariant as follows:

$$\textit{eating} \Rightarrow \textit{left\_fork} \wedge \textit{right\_fork}$$

This says that whenever the philosopher is eating, he requires access to the two universes referenced by the variables *left_fork* and *right_fork*. As long as eating is *false*, the philosopher does not require access to universes *left_fork* and *right_fork*. But as soon as eating becomes *true*, both *left_fork* and *right_fork* must be true for the entire formula to be *true.* Consequently, the process must acquire both universes before it can proceed.

When referenced within a realm constraint, a universe variable evaluates to *true* if and only if the universe is a member of the realm. Basically, *u* is a shortcut for a pseudo-expression *u.is_available.* Note that universe invariant specifications are not subject to the same access restrictions as processes. Unlike a process, a universe invariant can reference the condition variables from any universe. This is because, like garbage collectors [11], the underlying synchronization engine, which evaluates these constraints, works transparently and can assume a globally consistent view.

## 3.3 Semantics: Scheduling = Satisfiability

We now describe the semantic interpretation of universe invariants and the construction of realm constraints. Observe that in a correct schedule, the invariants of each universe in the realm of each process must be simultaneously satisfied. Using this observation, we devised an algorithm for computing the realm of a process dynamically. Essentially, a realm includes a minimal set of universes such that the conjunction of all the universe invariants is true. We call this conjunction of universe invariants the *realm constraint*.

Our run-time system bases scheduling on the satisfaction of realm constraints, allowing a process to proceed only if progress does not violate correctness. More precisely:

– If the constraint of the realm of a process is *forever unsatisfiable*, then the program is incorrect. This condition results either from a deadlock or a constraint that is inherently unsatisfiable.

– If the constraint of the realm of a process is *currently satisfiable*, then the process may proceed. The run-time system will allow the process allocate the required universes and then to continue execution.

– If the constraint of the realm of a process is *currently unsatisfiable*, then the process cannot proceed. The run-time system blocks the process from running until the constraint again becomes satisfiable.

To illustrate this decision process at work, consider the familiar producer/consumer example. The consumer universe will have the following invariant:

$$\textit{reading} \Rightarrow \textit{queue.has\_data} \wedge \textit{queue}$$

Here, *reading* and *queue.has_data* are both condition variables, *queue* denotes the queue universe. The formula enforces the following behavior: While *reading* is *false*, nothing is required. However, if *reading* becomes *true*, then both *queue.has_data* and *queue* have to become true for the process to proceed. In particular, this enforces idle waiting (blocking until the queue contains data that can be read) and also acquires the queue universe. So, as soon as the process proceeds it knows that it has access to the queue and that the queue has data for it to read.

Observe that with this constraint in place, the following code is correct:

```
if queue.has_data then
  item := queue.head
  queue.remove
end
```

Because the system guarantees the invariance of the constraint, as soon as the condition *reading* is true, the above code is automatically correct. If we made a mistake in specifying the universe invariant, then we would get a runtime error as soon as we tried to access an object outside the current realm.

Unfortunately, unlike deadlocks, starvation is not a correctness condition that can be be expressed in terms of universe constraints. If the scheduler were to not consider the possibility, a correct program could still exhibit starvation problems. As a result, the scheduler needs to implement an algorithm such as the one described in [7] to handle the acquisition of multiple locks fairly.

## 4. CONCURRENCY AND COMPOSABILITY

The universe model enriches module interfaces with constraints that relate the availability of a service to the *status* of the object that provides the service. At run-time, a realm will only contain universes such that the conjunction of constraints for each universe in the realm is true. The run-time system migrates universes among realms in order to satisfy these constraints. Consequently, the constraints implicitly describe the allowable sequencing of services, and they do so in a way that is dynamically extensible. This section describes the need for the dynamic interpretation of sequencing constraints and illustrates how the universe model accommodates this need.

Dynamic derivation of module-service sequencing constraints is important because the information may not be statically derivable *without violating the principle of information hiding.* Consider, for example, the following case: `ob` requires exclusive access to another object `cache` to properly complete both `query()` and `action()`. Furthermore, this cache object is also accessed by other objects and could be changed between the calls of `query()` and `action()`. If clients of `ob` are unaware of this dependency, then there is no way to know that `cache` must be locked for the duration of the transaction.

To inform clients of such a dependency in a traditional module interface, requires exposing implementation-dependent information to client programs—a violation of the principle of information hiding. On the other hand, if this information is buried inside the body of a procedure, then client programs will be unaware of the dependency. The universe model makes what we believe is a reasonable compromise between the separation of concern afforded by traditional information hiding and the design problems that arise by extending a module's interface with status and service-constraint information in an ad-hoc way. Specifically:

1. Concurrency constraints are analogous to class invariants: They are minimal descriptions of correct behavior of a component; and
2. Realm computation is performed transitively—as a closure of universes—based on the need to satisfy universe constraints. After performing this closure, a realm will include all the universes that the process must access for the current transaction.

To understand how these features accommodate dynamic extension without compromising the benefits of separation of concerns, recall the dining philosophers example. The universe constraint for a single philosopher simply states:

$$\textit{eating} \Rightarrow \textit{left\_fork} \wedge \textit{right\_fork}$$

No mention is made about any details regarding the universes *left_fork* and *right_fork*. The *philosopher* universe does not need this information to function properly. Of course, the *left_fork* universe may include a reference to a *dish* universe, or others. But this is non-local information, and the *philosopher* universe does not need to know about it. It will, however, still result in *dish* being included in the realm of the process when necessary.

Of course, enriching a modular interface with semantic information places constraints on the run-time system. But concurrency constraints are abstract and declarative, an improvement over having *concrete* implementation details exposed. Furthermore, in most cases other components are not going to even rely on these constraints; as with class invariants, their use is mainly internal as well as to guide the scheduling algorithm. Concurrency constraints add a semantic "glue" that allows us to connect universes transparently, without exposing more information than necessary.

## 5. CASE STUDY

In this section, we discuss a specific concurrent system: a university-enrollment server for use by students and instructors. While somewhat simplified, it is still sufficiently complex to illustrate the viability of our approach. Section 5.1 discusses the structure of the implementation. Section 5.2 uses the example to explain the abstract concepts from section 3. Finally, Section 5.3 shows how composition is supported in our example.

### 5.1 Enrollment Example

Our example focuses on the part of the enrollment system that allows students to log in and enroll in or drop a course. Universes are objects created as instances of a class, which must be derived from the system class UNIVERSE. Our example uses several universe classes: LOGIN handles the login process; STUDENT_TRANSACTIONS controls enrollment or dropping on a per-student basis; and COURSE is an abstract class whose subclasses access course-related data, such as number of students in the course. The example also requires two auxiliary classes: AUTHENTICATION to manage passwords; and ACCOUNTING to process billing and registration holds.

The universe class LOGIN handles the login and authentication process. Initially, each process owns a distinct LOGIN universe and nothing else. The following is part of class LOGIN's interface (without methods and unimportant attributes).

```
class LOGIN < UNIVERSE is
  student: STUDENT_TRANSACTION;
  authentication: AUTHENTICATION;
  logging_in: BOOLEAN;
concurrency
  logging_in ⇒ authentication;
  ¬logging_in ⇒ student;
end
```

This declares class LOGIN as a subclass of UNIVERSE[1] By inheriting from UNIVERSE, we both acquire the concurrency tools provided as methods by that class and declare the class to be of a type that allows instances to be assigned to a universe variable. Login connects to two other universes. A student universe that refers to the universe that holds the data of the student that logged in; and a central authentication universe that holds all the passwords and other data needed to verify the authenticity of the login.

Naturally, we require access to the authentication universe only during the login process, and access to the student universe only once we're done with the login process. At some point, the following statement will be executed:

$$\textit{logging\_in} := \textit{false};$$

at which point, the system will continue to process the student.

For brevity, we do not include details of the authentication process, which is unrelated to the topic of this paper. However, the STUDENT_TRANSACTION universe class proves to be more interesting:

```
class STUDENT_TRANSACTION < UNIVERSE is
  added_course, dropped_course: COURSE;
  accounting: ACCOUNTING;
  swapping, dropping, adding: BOOLEAN;
```

[1]Our inheritance notation is a hybrid of Sather [15] and Eiffel [14].

```
concurrency
    $swapping \vee dropping \Rightarrow dropped\_course$;
    $swapping \vee adding \Rightarrow added\_course$;
    $swapping \vee adding \vee dropping \Rightarrow accounting$;
end
```

Observe that various conditions can require the inclusion of a course universe. Both swapping and dropping a course require that we disenroll the student from a certain course; likewise, both swapping and adding a course require that the student be enrolled; furthermore, when swapping courses, this should all occur atomically. The above conditions take care of this. Finally, checking and updating accounting information (such as holds) will be necessary as part of the same transaction, no matter what the transaction is.

It is important to note that the implementation needs a construct to simultaneously assign values to several variables at once. We assume that the language provides a multi-assignment facility, such as:

$swapping, dropping, adding := true, false, false;$

A multi-assignment primitive is necessary: Simulating it with a sequence of single assignment statements would result in unnecessary calls to the scheduler and might cause deadlock.

Next, we consider how to access the course. For the purpose of enrolling, the course *and* all of its prerequisite courses must be acquired atomically. Otherwise, if there were more than one prerequisite, two concurrent enrollment operations might acquire one prerequisite each and deadlock afterwards while trying to acquire the other prerequisite (see figure 3—course 1 has acquired prerequisite 1, course 2 has acquired prerequisite 2, but neither can access the other one). On the other hand, dropping a course requires only the course object itself and not the prerequisites. In order to accommodate both of these uses, we will use front-end objects to access a course object. These front-end classes derived from the same abstract class as the courses themselves, but with the additional logic to handle prerequisites.

When the user is dropping a course, there is no need to check prerequisites. The front-end in this case will be a simple universe object that delegates all operations to the real course object. Such a delegate is declared as follows:

```
class STANDALONE_FRONTEND < COURSE is
    real_course: COURSE;
concurrency
    real_course;
end
```

For other purposes, such as enrolling in a course or swapping courses, the front-end must also reference the prerequisites. Moreover, all of the prerequisites must be acquired in one atomic action, as discussed above. This is accomplished as follows:

```
class ENROLLMENT_FRONTEND < COURSE is
    real_course: COURSE;
    prerequisite1, prerequisite2: COURSE;
    prerequisites: INTEGER;[1]
```

[1]For the purpose of this paper, we keep the notation simple and do not include the **forall** and **exists** operators that can express constraints over lists of universes. As an unfortunate side effect, this limits us to a finite number of prerequisites.

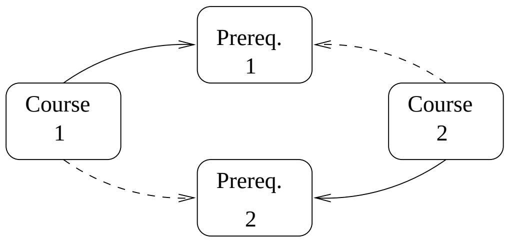


**Figure 3: A Deadlock Situation**

```
concurrency
    real_course;
    $prerequisites \geq 1 \Rightarrow prerequisite1$;
    $prerequisites \geq 2 \Rightarrow prerequisite2$;
end
```

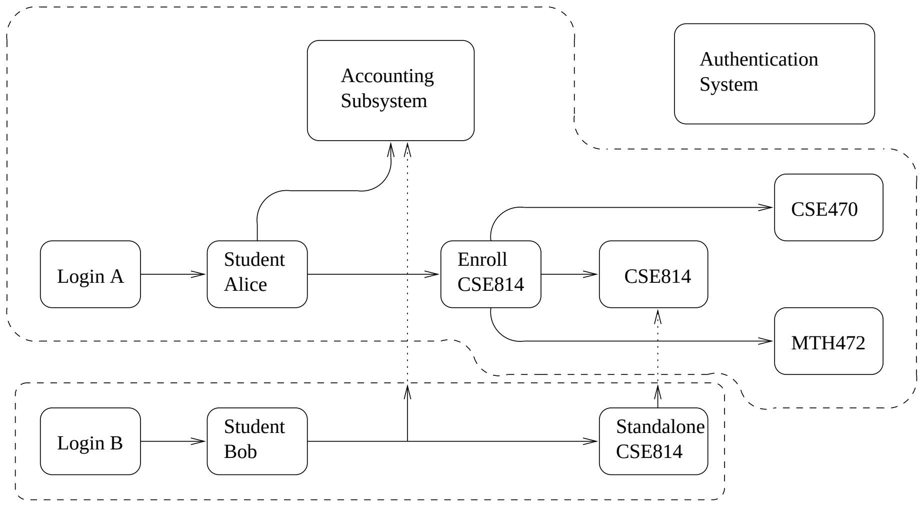
For instance, if course *cse814* requires both *cse470* and *mth472* as prerequisites, then the corresponding variables would hold references to these courses, and $prerequisites = 2$.

## 5.2 Stepping Through a Sample Transaction

Figure 4 shows us a snapshot of the system in progress. Two students—Alice and Bob—are currently logged in. Alice is enrolling in *cse814*, while Bob is trying to drop the same course. Two different processes are operating: One originates in the login universe for Alice; whereas the other originates in the login universe for Bob. The dashed lines surround the realm for each process.

Consider Alice's process. We are past authentication, so $logging\_in = false$. The *student* variable refers to Alice's student universe. Because Alice is in the process of enrolling in *cse814*, we are referencing both the accounting subsystem and the enrollment frontend for the course itself, and according to the concurrency constraints, both universes are included in our realm. The enrollment frontend for *cse814*, in turn, requires the course and its two prerequisites.

Bob's process is currently blocked, waiting for Alice's process to release the accounting universe and *cse814*. If Bob's process were to proceed, a call would be made to a method outside the current realm, resulting in a runtime error. Signaling a runtime error is necessary, since otherwise both Alice's and Bob's process would simultaneously operate on the *cse814* universe, with possibly disastrous consequences. As soon as Alice finishes enrolling in *cse814*, the variable *adding* in her student universe will become false. The variables *dropping* and *swapping* were hitherto false anyway. Consequently, the accounting universe and *cse814* become available. Because the scheduler intercepts the assignment to *adding*, it knows to try to schedule Bob's process again. At this point, both the *Accounting Subsystem* and *cse814* will become part of the realm of Bob's process. The prerequisite courses, on the other hand, will remain available for other operations.



The Enrollment system during execution. Two threads exist, originating in universes *Login A* and *Login B*.

- Boxes denote universes. Dashed lines surround realms. Solid lines ending in arrows indicate that a concurrency constraint in the originating universe requires the inclusion of another universe and that it is thus part of the realm. Dotted lines indicate the same, except that at the moment the inclusion cannot be completed and thus blocks the process.
- *Alice* is currently enrolling in *cse814*. *Bob* would like to drop *cse814*, but this cannot be done until *Alice* relinquishes control of the accounting subsystem and the *cse814* universe.
- The authentication universe is currently not part of *any* realm—we are past the login phase for all active connections, and thus we don't access to it.

**Figure 4: Full Enrollment System**

Depending on the planned action, we need access to either a course to be dropped, a course to be added, or both. In particular, whenever a student plans to drop one course and enroll for another instead, then this transaction needs to be atomic—otherwise a student might drop one course, be unable to enroll for the substitute, and also unable to re-enroll for the original course.

The simple assignment *swapping* := *true* will then guarantee that the transaction cleanly takes place, because this will automatically ensure that both the course to be dropped and the one to be added will be acquired simultaneously.

Some classes, such as EnrollCSE814, *relay* concurrency constraints. We anticipate that *relay universes* will become a common programming idiom in this model. Consider, for instance, that course *cse814* always required both course *cse470* and *mth472*. Then the concurrency constraint for *cse814* would contain the following clause:

$$cse470 \wedge mth472$$

No process normally originates in *cse814*. It is usually only required when a student is adding or dropping a course, or an instructor modifying course information. However, enrollment in *cse814* requires checking that the student has taken both *cse470* and *mth742*. So, whenever universe *cse814* is acquired, then its concurrency constraint must be satisfied, thus forcing inclusion of both *cse470* and *mth472*. Of course, this process is applied transitively for the concurrency constraints of the prerequisite courses until all of them are included.

Unfortunately, this makes some simplifying assumptions. For instance, it assumes that the prerequisite relation is fixed. Changing prerequisites requires recompiling the program. Similarly, dropping does not require the locking of all dependent universes. Such a situation can still be handled, of course, but we may need auxiliary code to avoid deadlock. This is why we included specific auxiliary "frontend" universes to deal with this case. The general idea of *relaying* inclusion requirements remains the same, of course, as can be seen in either of the frontend universes.

## 5.3 Composition in the Enrollment Example

The example illustrates how we manage concurrency in the presence of complex sets of universes with simple, obvious and completely localized concurrency constraints. The frontend universe for *cse814* illustrates how the scheduling mechanism transcends layers while still maintaining information hiding. If we look at both the class interface and the diagram, we notice that the student universe doesn't have to rely on any data related to *cse814*; all of that is encapsulated in the frontend.

While this in itself would not be very surprising for a sequential system, maintaining this transparency in the presence of concurrency has been shown to be trickier. Observe that a programmer using our model need not be concerned with locking objects in the right order. She simply specifies relations between related objects, and the scheduling mechanism will compute the closure of all of them, deriving the global predicate that describes our set of components from purely local information.

A monitor-based solution would be much more complicated. As we observed in section 2.3, we would have to expose an undesirable amount of information about the components involved in the transaction. In this particular example, the STUDENT_TRANSACTION class would have to be informed about all the prerequisites. A way around this would be to include a multi-object locking mechanism in the enrollment frontend—this way we would be getting almost the same results. We would then enclose the enrollment transaction in calls that lock and unlock this set of objects. However, this still would not deal with our need to acquire a lock on the accounting system at the same time, and it would not accommodate additional layers. The one big advantage of the universe model with respect to these problems is that we can describe the **entire** set of components, no matter how large and involved with just a set of fully local predicates.

## 6. DISCUSSION AND RELATED WORK

We now discuss our model in relation to others based on a number of criteria for evaluating the efficacy of a concurrency mechanism.

### 6.1 Security

A language is deemed secure if common programming errors are cheaply detectable at run time and not allowed to give rise to machine-dependent effects, which are inexplicable in terms of the language itself [9]. There are many imperative languages in use without a secure concurrency mechanism. Java in particular has been criticized [5] for the limited safety of its native concurrency mechanism, but some others do not even have the luxury of an existing implementation of concurrency (e. g., Eiffel).

Security requires a certain degree of redundancy. For example, whereas the upper bound of an array is not required to perform an indexing operation; the additional information provided by array bounds allows for checking the validaty of an operation at run time. In some cases, errors can be detected at compile time; specifically, most statically typed languages will allow you to eliminate most type errors at compile time, due to the additional information introduced by the type system.

The main problem that any concurrent imperative language faces is to ascertain that mutual exclusion works properly. And while there is no shortage of approaches that replace the basic concurrency mechanisms of old with more sophisticated variants, few of them introduce the necessary redundancy that can aid in the discovery of sharing violations. One approach is the SCOOP mechanism described in [14], which requires that an object belonging to a different process be locked before it can be accessed. However, this requirement only works for transactions contained within a single method; it fails for hierarchical systems and composite transactions that are not localized within a single method. Synchronizers [8] allow the programmer to attach constraints to methods, but there is no guarantee for the constraints to be exhaustive. While not as easily foiled as traditional monitors, it is still fairly easy for undesired process interaction to occur.

The universe model is fairly unique in that it provides an unprecedented amount of redundancy. Conceptually, it is very similar to the checking of array bounds. The concurrency constraints indirectly describe the set of universes (and thus objects) a process is allowed to access, just as array bounds describe the set of memory cells that the indexing operation may access. Any access beyond the bounds of this set can then be trapped and turned into a runtime error. This usually happens for one of two reasons. Either the process accesses an object that it does not require. In this case, the code itself is at fault. Or the process accesses an object that it needs, but which according to the concurrency constraint is not available. In this case, the specification is wrong.

### 6.2 Separation of Concurrency and Computation

It is desirable—especially for the design of large systems—to separate the pure computational parts of a system from the parts that control concurrent interaction, ideally in a way that does not hinder power or expressiveness [1]. Computation and concurrency specifications should be expressible in their own notations, and they should be easily composed. We view this as a classic multi-paradigm specification problem [21]. In this view, partial specifications can be composed if there is some shared vocabulary (usually a set of common variables) using which one partial specification may influence or be influenced by another [22]. In the universe model, this shared vocabulary is the set of condition variables of all of the universes in a system. In other approaches, most notably coordination languages, the shared vocabulary is implemented by other means. We believe that the universe model fares better with regard to the interleaving of computation and concurrency than do these other approaches.

One of the older and more popular approaches to modeling concurrent interaction (especially in distributed, heterogeneous systems) is the Linda model [6]. Linda primitives can be added easily to most existing languages. However, we also observe some drawbacks. Consider the following implementation[1] of the Dining Philosophers problem:

```
void philosopher(int i)
{
  for (;;)
  {
    think();
    in_and(("fork", i), ("fork", (i+1)%5);
    eat();
    out("fork", i);
    out("fork", (i+1)%5);
  }
}
```

This is superficially similar to an equivalent design using the universe model:

[1]Traditionally, Linda doesn't have a primitive like in_and to describe the atomic acquisition of multiple tuples. We use it to make the example more compact and more easily understandable.

```
loop
  think;
  eating := true;
  eat;
  eating := false;
end

concurrency
  $eating \Rightarrow left\_fork \wedge right\_fork$
```

However, there are also a few critical differences. In particular, the abstract, descriptive assignments to *eating* become explicit concurrency control statements in Linda, and more importantly, they refer to implementation details regarding the topology of forks rather than the abstract status of the computation (i.e., thinking vs. eating). This results in an interleaving of computation and concurrency control statements which appears to be fairly typical of approaches that operate via a shared blackboard or tuplespace. An obvious concern is that this interleaving—driving the computation by changes to a central data structure—might result in maintenance problems insofar as changes to the computational part might require concomitant changes in the part that controls concurrent interaction, and vice versa.

By contrast, condition variables do not refer to a universe's implementation details, but instead refer to the abstract state of a universe's computation, which is often part of the universe's functional specification. Here, we distinguish the *concrete state* of a universe from the *abstract state*, which is a more concise representation that conveys the computation status in order for an external agent to properly coordinate execution with other universes. By using condition variables to encode the abstract state, assignments to condition variables announce changes in the concurrency status of a computation using a shared vocabulary with the concurrency specification. While possibly unnecessary for the actual computation, the very fact that condition variables should describe the abstract state makes it easy to add them to existing code. By and large, we can even augment code in this way without being aware of how concurrency constraints deal with them. Conversely, concurrency constraints rely solely on the abstract state encoded in condition variables without requiring details of the implementation. The abstract state therefore functions as an insulating layer that separates computation and concurrency control.

It should be noted that unlike many other approaches, we chose to abstract out the state, but not the transitions between states (as opposed to Lotos [4], CSP [10], etc.). This is because we suspect that focusing on state allows for a design that exhibits more stability under change. There are some classical problems in concurrent system design (such as the so-called inheritance anomaly [13]) where small changes in the design result in small changes in the abstract statespace, but huge changes in the transition system.

## 6.3 Accessibility

An important aspect of every language, construct, or tool is that it is easily accessible to the average software developer. Difficulty to comprehend an idea not only hinders popularity (and thus acceptance) of advanced mechanisms, but also adds to the time and expenses for training and can be another source of errors. In most cases, companies that develop software are reluctant to take big leaps, relying on piecemeal engineering instead. This has been made evident by the acceptance of Java and C++, whose popularity is largely based on the assumption that they are close to C (whether appearances are deceiving in either case is another matter, and largely irrelevant). An example from concurrent programming is Linda [6], which can easily be added to existing languages, and which is also easily understood. As a matter of fact, the concept of a tuplespace to coordinate parallel processes has become extremely popular due to the almost unrivaled ease with which the concept can be understood and used (see, for instance, the JavaSpaces variant [19]).

When designing the universe model, we considered this aspect very carefully; we already knew that mechanisms such as Design by Contract[1] [14], while not as powerful as full-fledged specification mechanisms, have been easily adopted by programmers and domain experts alike. In our design, we sought to mirror the successful elements of Design by Contract. Conceptually, our concurrency constraints are very similar to class invariants. In fact, as we established above, they describe the correct behaviour of the universe class with respect to concurrent interaction. In addition, the constraints do not require understanding of advanced constructs, as they are for the most part formulas from propositional logic, which are easily understood by most programmers. We do add the additional complexity level of satisfiability, but expect that the allocation of universes *by need* as defined by the concurrency constraint is no harder to grasp than, say, unification in Prolog.

## 6.4 Efficiency

The efficiency of a concurrent programming language or mechanism is affected by two parameters—frequency of context switches to invoke the scheduler, and the granularity of concurrency. At first glance, it may appear that the universe model would fare poorly on both fronts. Whenever condition variables change, considerable reevaluation of concurrency constraints may be required. We observe in [2] that universe constraints actually use only a subset of full propositional logic. Moreover, when evaluating a universe constraint, the values of condition variables are fixed. For example, when considering the following clause from previous examples, only the underlined part needs to be satisfied:

$$eating \Rightarrow \underline{left\_fork \wedge right\_fork}$$

We show in [2] that we only need conjunction and disjunction, plus potentially the implication operator ( $\Rightarrow$ ) to deal with all such situations. In fact, most of the examples we have modeled do not even require the latter, and can convert all clauses that are to be checked for satisfiability to disjunctive normal form prior to execution. In this form, satisfiability becomes trivial—we simply find a satisfying assignment to one of the subclauses. Computing the closure over many universes is slightly more complicated, but still requires only the addition of further subclauses of the disjunctive normal form.

[1]Design by Contract allows for the specification of preconditions and postconditions of methods as well as class invariants, as long as they are executable. They are then checked at runtime. While not as powerful as a full specification, the added redundancy allows for additional error-checking and provided documentation that by its very nature does not tend to diverge from the actual code.

An implementation that encodes these subclauses as bit-vectors should be sufficiently fast (except that distributed systems with high communication latency will require a more sophisticated solution). We are currently extending our implementation to validate this hypothesis.

The issue of granularity is more of a problem; specifically, the universe model is *not* suited (for instance) to deal with highly parallel operations on large matrices, as it is designed around the encapsulation boundaries of object-oriented systems and should not be expected to compete with algorithms that are optimized for numerical operations (strip-mining, etc.).

In general, it can be said that the universe model is designed with structural and security concerns in mind, not with speed. However, that does not mean that we are willing to sacrifice efficiency. For "everyday" applications such as web-servers, e-commerce, and accounting the model should work just fine, and it has been designed so that speed does *not* degrade compared to other solutions. What the system intentionally does not do is to utilize the resources of highly parallel hardware, perhaps even with context switching at the instruction level.

## 7. REFERENCES

[1] F. Arbab and G. A. Papadopoulos. Coordination Models and Languages. In *The Engineering of Large Systems.* Advances in Computers. Academic Press, 1998.

[2] R. Behrends and R. E. K. Stirewalt. A High-Level Approach to Concurrency. Tech. Rep. MSU-CSE-00-6 (March 2000), Department of Computer Science and Engineering, Michigan State University.

[3] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, New York, 1990.

[4] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language Lotos. *Comp. Netw. ISDN Sys.* **14** (1) (1987).

[5] P. Brinch Hansen. Java's Insecure Parallelism. *ACM SIGPLAN Notices* **34** (4), 38-45 (1999).

[6] N. Carriero and D. Gelernter. Coordination languages and their significance. *Communications of the ACM* **35** (2), 97–107 (1992).

[7] C. M. Fleiner and M. Philippsen. Fair Multi-Branch Locking of Several Locks. In *International Conference on Parallel and Distributed Computing and Systems*, pages 537–545, Washington D.C., October 1997.

[8] G. Agha and S. Frølund. A Language Framework for Multi-Object Coordination. In O. Nierstrasz, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, pages 346-360. LNCS 707. Springer-Verlag, 1993.

[9] C. A. R. Hoare. Hints on Programming Language Design. In C. J. Bunyan (Ed.), *State of the Art Report 20: Computer Systems Reliability*, pages 505–534. Pergamon/Infotech Publishing, 1974.

[10] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[11] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[12] S. Letovsky and E. Soloway. Delocalized Plans and Program Comprehension. *IEEE Softw.* **3** (3) (1986).

[13] S. Matsuoka and A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In G. Agha and P. Wegner and A. Yonezawa, *Research Directions in Concurrent Object-Oriented Programming*, pages 107-150. MIT Press, Cambridge, MA, 1993.

[14] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1997.

[15] S. M. Omohundro. The Sather Language. Tech. Rep. (1991), International Computer Science Institute, 1947 Center Street, Suite 600, Berkely, California 94704.

[16] C. Rich and R. C. Waters. *The Programmer's Apprentice.* Addison Wesley, 1990.

[17] S. Rugaber and R. E. K. Stirewalt and L. Wills. Understanding Interleaved Code. *Journal of Automated Software Engineering* **3** (1) (1996).

[18] S. Rugaber and R. E. K. Stirewalt and L. Wills. The Interleaving Problem in Program Understanding. In *Proceedings of the IEEE Second Working Conference on Reverse Engineering*, 1995.

[19] A. Shah. State of the Art: JavaSpaces. *Java Report: The Source for Java Development* **2** (5), 16 (1997).

[20] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, New Jersey, 1993.

[21] P. Zave. A Compositional Approach to Multiparadigm Programming. *IEEE Software* **6** (5) (1989).

[22] P. Zave and M. Jackson. Conjunction as Composition. *ACM Trans. Softw. Eng. Meth.* **2** (4), 371-411 (1993).