Dynamic Synchronization – A Synchronization Model through Behavioral Reflection

Jorge Ressia & Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch

ABSTRACT

In conventional software applications, synchronization code is typically interspersed with functional code, thereby impacting understandability and maintainability of the code base. At the same time, the synchronization defined statically in the code is not capable of adapting to different runtime situations. We propose a new approach to synchronization which strictly separates the functional code from the synchronization requirements to be used and which adapts objects to be synchronized dynamically to their environment. First-class *synchronization specifications* express safety requirements, and a *Dynamic Synchronization System* dynamically adapts objects to different runtime situations. We present an overview of a prototype of our approach together with several classical concurrency problems, and we discuss open issues for further research.

Categories and Subject Descriptors

D.3.2 [**Programming languages**]: Smalltalk; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.1 [**Software**]: Programming Techniques—*Object-Oriented Programming*; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Context, Concurrency, Reflection

Keywords

Context-oriented Programming, software composition, behavioral variation, reflection, context

1. MOTIVATING EXAMPLE

Pier [10] is a fully object-oriented content management system implemented in Smalltalk which has a large number of users. Pier provides the mechanisms to create, edit and manage hypertext pages on the web. Pier uses the Command pattern [6] in order to model the different actions that can be performed by the user like editing, adding, moving and copying pages. Each command might affect a different number of pages depending on its semantics. For instance, the *edit command* will only modify one page, the page being edited, while the *move command* will affect the page itself, all its children, the origin parent page and the destination page.

The fact that different commands touch quite different sets of pages poses a non-trivial *dynamic synchronization problem*. If two clients trigger two commands and the sets of pages that they modify overlap then we will have a data race. Two commands that might modify at least one shared page should be executed mutually exclusively. Defining a mutual exclusion mechanism is not a trivial task since the number of pages targeted by commands can vary. The developers of Pier decided to attack this problem conservatively. A global mutex is used to synchronize all commands, which means that if there are two commands whose page interests do not overlap, one of them will have to wait for the other to complete, even though there is no conflict. We seek a solution to the dynamic synchronization problem which is safe, yet optimally live. Since one cannot know *a priori* which objects may be modified by a given thread, we need a solution that dynamically adapts itself to the interests of the actually running threads.

We define the *interests* of a thread to be the set of objects for which that thread may require read or write access. We propose a solution to the dynamic synchronization problem that dynamically computes the interests of different running threads. In a nutshell, one simply specifies a synchronization specification, which declares which messages may potentially conflict, and specifies a means to compute the thread interests. This information is then used at runtime to adapt the synchronization policy. If thread interests overlap, then a mutual exclusion policy, for example, can be imposed, whereas if no conflicts exist, then no synchronization is required.

In our prototype, we make use of the Reflectivity framework [3] to dynamically adapt the behavior of clients and resources. The decision about how a particular situation should be synchronized depends on the interests each running thread has.

2. SOLUTION: DYNAMIC SYNCHRONIZATION

In order to improve the way Pier manages concurrent execution of commands we will present how this can be achieved with our approach. In conventional software appli-

IWST'09 August 31, 2009, Brest, France.

cations, synchronization code is typically interspersed with functional code, thereby impacting understandability and maintainability of the code base. Our approach also tackles this issue by providing abstractions which strictly separate the functional code from the synchronization policy to be used and dynamic adaptation to the application environment.

We introduce the notion of a first-class *synchronization specification*, which expresses the safety requirements for a set of collaborations. A synchronization specification decouples the synchronization requirements from the actual policy to be used. At runtime the synchronization specification is used to dynamically select the most suitable policy to put into place.

We can specify these requirements for the *add command* as follows:

```
SynchronizationSpecification
    for: PRAddCommand
    on: #validateAndExecute
    interestedIn: #pagesPotentiallyModified
```

This specifies a synchronization constraint for instances of PRAddCommand. Whenever the message validateAndExecute is sent concurrently to two or more instances, a potential data race exists. The message pagesPotentiallyModified can be sent to the PRAddCommand instances to determine which actual pages may be accessed by the command. This information can then be used by the runtime system to adapt the synchronization policy.

It is also possible to specify a set of messages that may need to be synchronized, and instead of sending a message to compute the thread interests, an arbitrary one-argument block (*i.e.,* an anonymous function) may be specified, taking the instance as its parameter.

We could also specify synchronization for the PRMoveCommand as follows:

```
SynchronizationSpecification
    for: PRMoveCommand
    on: #validateAndExecute
    interestedIn: #pagesPotentiallyModified
```

As we can see the specifications for these two commands are almost identical. We can instead provide the following specification for all commands:

```
SynchronizationSpecification
    forAllSubclassesOf: PRCommand.
    on: #validateAndExecute
    interestedIn: #pagesPotentiallyModified
```

This specification models the synchronization requirements for command execution in Pier. In order for this specification to be applied to the running application it has to be registered with the *Dynamic Synchronization System* (DSS). This system is responsible for adapting the behavior of objects at runtime according to the registered specifications. Adapted code delegates to the DSS the decision which synchronization policy should be put in place at runtime, depending on the interests of the running threads. This cannot be another source of race conditions since the access to the DSS is synchronized.

After the specification is registered, every time a command receives the message validateAndExecute the adapted code will be executed. This code will delegate to the DSS the responsibility of deciding which synchronization policy to use. The DSS will determine the interests of running threads and impose mutual exclusion in case of a conflict. If there are no overlapping interests, the threads will run to completion without interfering between each other. Further details are given in Section 3.2.

3. DETAILS

In this section we present the implementation details of our application.

3.1 Code Adaptation

Code is adapted by the DSS using the AST rewriting facilities of the Reflectivity framework [3]. All methods listed in a synchronization specification are adapted. The adaptation process adds synchronization code which collaborates with the DSS to allow the objects to dynamically adapt their synchronization policy. Our prototype currently only provides one policy which is mutual exclusion.

By separating the synchronization code from the functional code we have also separated the requirements from the implementation. The DSS can be satisfied using different approaches. We have selected a mutex approach but the adaptation technique does not limit the use of other approaches like transactions, monitors, *etc.*

3.2 Synchronization mechanism

In order to achieve mutual exclusion between the behavior of different objects we chose a standard mechanism. A unique mutex is assigned to every object in the thread interests. These mutexes are ordered in a global fashion. The adapted code delegates to the DSS the calculation of the interests and the construction of a mutex. This mutex could be a set of ordered mutexes or a single mutex, depending on the number of objects in the thread interests.

```
message
  (DynamicSynchronizationSystem current
    mutexFor: self
    in: aSpecificationId)
      critical: [ ∧ originalMessageBody ]'
```

As we can see in the previous adapted code, the decision which mutex or set of mutexes to use is delegated to the DSS. The adapted object is passed as an external collaborator. The specification id is the unique identification provided by the DSS when the specification was registered. The specification id is required since the DSS uses it to identify the synchronization specification which was used to adapt the code thus accessing the block for calculating the interests at runtime. Finally the original message body will be executed inside the critical message of the mutex provided.

3.3 Example

We use a classic, academic problem to illustrate the approach.

In the Dining Philosophers problem several philosophers spend their time thinking and eating. In order to think they first need to eat. In order to eat each of them needs two forks. There are not enough forks for all of the philosophers to eat at the same time.

In Figure 2 we show one possible solution for this problem. Each philosopher has two main actions, eat and think. When a philosopher first tries to eat he must pick up both

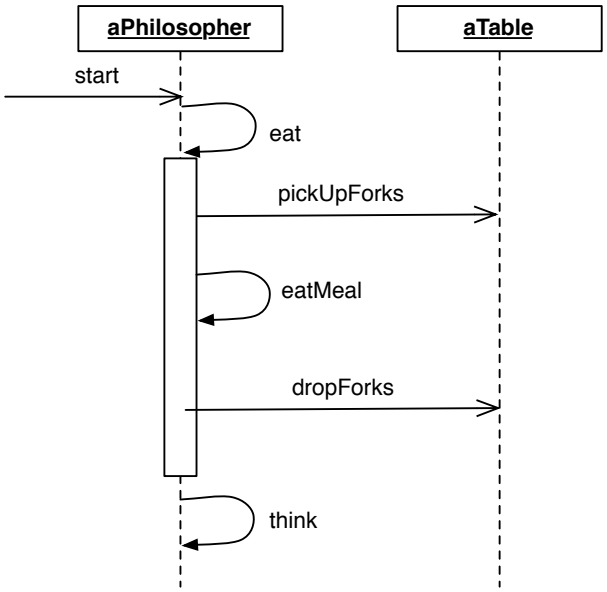

Figure 1: Dining Philosophers sequence diagram

forks in front of him. After that he can actually eat, and afterward drop the forks back on the table.

If we would ask someone to express in ordinary words the synchronization problem of this example he would say that philosophers need forks to eat, and two philosophers should be able to eat at the same time if they do not share any forks. Translating this to our framework, we would specify that when a philosopher wishes to eat, he is interested in forks.

```
SynchronizationSpecification
    for: Philosopher
    on: #eat
    interestedIn: #forks
```

The synchronization specification expresses this requirement declaratively. When trying to eat, a philosopher should be synchronized with respect to the forks he is interested in. The synchronization specification makes this requirement explicit and it helps the developer to step back and think the problem in at a higher level, closing the semantic gap between the problem domain and its modeling solution.

Given the situation shown in Figure 2, if Philosopher 1 is the first to start eating then he will hold forks 1 and 2. If then Philosopher 2 tries to start eating he will be blocked waiting for fork 2 held by Philosopher 1. When Philosopher 3 then tries to start eating he needs both forks 3 and 4, Since these are available, he will be able to eat. When Philosopher 1 stops eating and leaves the forks on the table Philosopher 2 will try to take forks 2 and 3. He will succeed taking fork 2 but depending on whether Philosopher 3 is done eating or not Philosopher 2 might need to wait for fork 3.

Every object which is declared to be a thread interest, in this case the forks, has an associated mutex used by the adapted code. The adapted code will send the critical: message to the mutexes always in the same order. This synchronization mechanism is safe since locking on mutexes in a consistent order for all threads avoids deadlock due to circular dependencies.

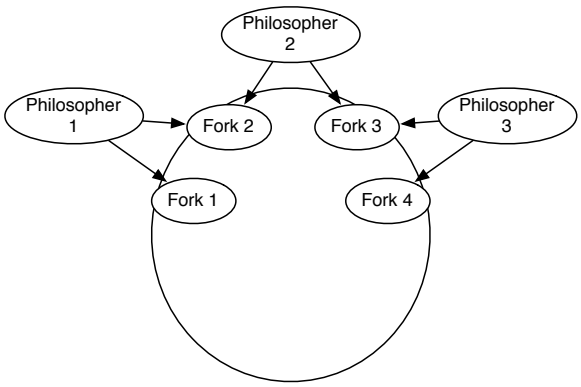

Figure 2: Philosophers Interaction

4. DISCUSSION

In this section we show some examples how synchronization specifications can help us think about concurrency from different perspectives.

4.1 Dynamic Adaptation

During the application life cycle the synchronization requirements can and do change, thus invalidating previous synchronization specifications. The synchronization specification abstraction addresses the problem of changing synchronization requirements as follows. Suppose that we slightly change Pier's Command specification by introducing a requirement which states that when the message validate is sent to a command it should be mutually exclusive with any other command validation and execution.

```
SynchronizationSpecification
        forAllSubclassesOf: PRCommand.
        on: #validateAndExecute
        interestedIn: #pagesPotentiallyModified
```

Suppose that a distracted developer would create a new specification for a new requirement which states that when the message validate is sent to an add command it should be mutually exclusive with any other command validation and execution. Instead of modifying the preexisting specification he defines a new one.

```
SynchronizationSpecification
        for: PRAddCommand
        onAll: #( validateAndExecute  validate )
        interestedIn: #pagesPotentiallyModified
```

We could introduce a rule in the DSS registration which states that if the objects belong to the same hierarchy, the blocks for evaluating the thread interests are the same and all the objects involved are polymorphic regarding the extra message validate; we are able to merge this too specifications as it is shown in Figure 3. When registering this new specification the DSS will detect this overlap and merge the two synchronization specifications. By reifying the specifications we are able to access the synchronization requirements at runtime. By no means is this a fixed behavior. On the contrary, this behavior can be changed when the DSS is created by providing a set of rules or at runtime by changing the set of rules. With this example we want to emphasize

how the synchronization specification abstraction assists us in thinking about our application from a higher level of abstraction regarding synchronization.

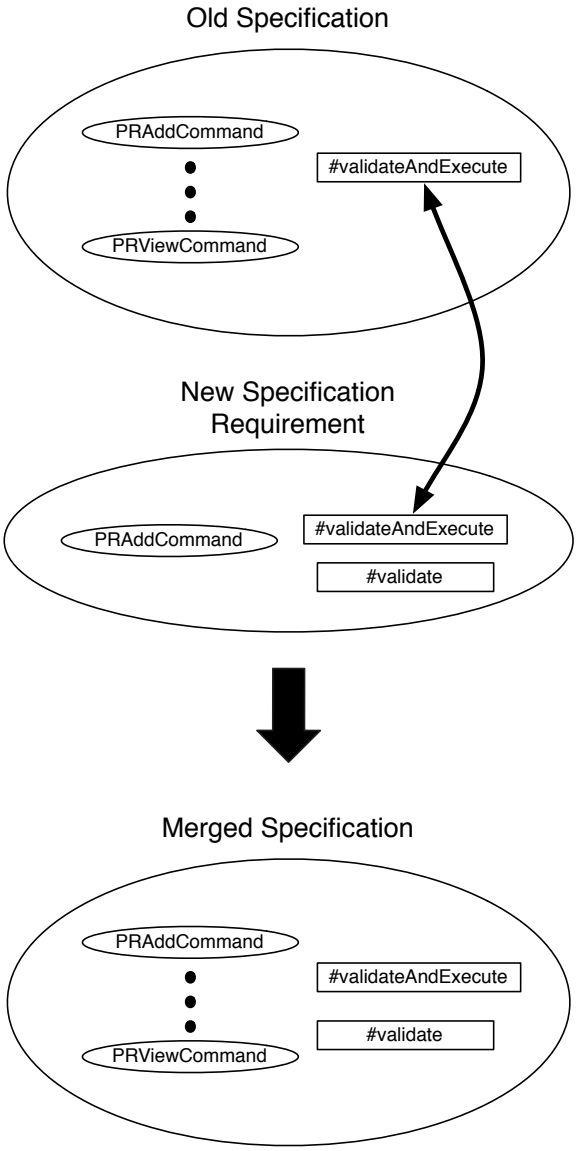

Figure 3: Specification with overlapping behavior

4.2 Configuration

Another adaptation strategy could be provided in order to change the way code is adapted. So far we have only experimented with a simple modification of this approach in which several objects are passed from the adapted code to the synchronization system in order to be used in the calculation of the thread interests. We have also experimented with different mutex providers. Instead of using the classical approach of assigning one mutex per object we modified this mechanism to consider groups of object. The group of objects modeled by a thread interest is assigned a single mutex. If afterwards, while these interests are active, another thread has overlapping interests then it will be assigned the same mutex and so on. With this approach we are saving the time consumed in sending the message critical: to a potentially large collection of mutexes. However, this approach has the drawback that in the worst case all threads might end up waiting on a single mutex even though they could execute safely. Suppose we have three threads each interested in two objects, (1 2), (2 3) and (3 4) respectively. Now suppose these threads run one after the other. Since their interests overlap, they will all end up waiting on the same mutex, even though the first and third threads have independent interests. In case of the dining philosophers, this strategy could lead to only a single philosopher being able to eat at a time.

These different strategies can be changed when the DSS is created by passing different collaborators to it. We plan to explore other synchronization strategies to provide the user with a wide variety of configuration scenarios.

5. PERFORMANCE

To assess the performance impact of our approach we have performed some benchmarks. The adapted code sends more messages than needed for traditional synchronization using semaphores in Smalltalk. We will analyze the specific slowdown related to this issue and then we will show results of slowdown in real problems.

We have compared the performance of a method which sends the critical: message to a semaphore against the adaptation of and empty message and for both of them we repeat its execution as it can be seen:

```
( 1 to: 1000000 ) do: [:each | anObject testSync ].
```

```
( 1 to: 1000000 ) do: [:each | anObject testAdapted ].
```

Table 1 shows the results when comparing a classical synchronization approach with our approach through adaptation. As expected the slowdown is considerable when no message is sent from the critical section. This implies that most of the time is invested in synchronization code. Since our approach adds message sends in order to determine which synchronization structure to use we obtain a substantial slowdown. If we now introduce empty message sends in the critical section we can observe that the slowdown is considerably smaller, down to around 30%.

It should be noted that the overhead observed for adapted code does not translate directly into a slowdown of a real world application using this technique. The overall slowdown depends on how often the adapted portions of the application are executed and how much of the application is adapted. If we take the Dining Philosophers problem and compare the slowdown between a classical approach and an adapted one we only get a 0.27% slowdown.

6. RELATED WORK

The idea of separating the problem domain code from the synchronization specific constructs was first pioneered by Lopes [9] in the D Framework. This work provides language constructions for mutual exclusion which have a similar behavior to the synchronization specifications. They also present language features for defining Coordinators which have the responsibility of coordinating the behavior of the threads. These language constructs use aspects in order to adapt the code.

	Classical (msecs)	Adapted (msecs)	Slowdown
0 message sends	117	2807	2300%
500 message sends	3524	6073	72%
1000 message sends	6849	9337	36%

Table 1: Performance Impact

Kienzle and Guerraoui [7] also address concurrency and failures from an aspect-oriented viewpoint. They use transactions to solve the problem and due to this the application code is still polluted with synchronization concerns. They demonstrate that although a certain level of separation of concerns is achievable some problems arise and they cannot be solved by aspectizing the transaction mechanisms. It is not possible to apply transaction aspects to previously non-transactional code due to the impossibility of automatically identifying irreversible actions.

Fabry *et al.* [4] introduced KALA, Kernel Aspect Language for Advanced transactions, a domain-specific aspect language for using advanced transaction management in a distributed system. This tool provides transactions as language constructs for dealing with concurrency and distributed issues. The main issue with this approach is that the problem domain code has to be modify to also contain the behavior related to transactional concerns.

Caromel *et al.* [2] introduce Sequential Objects Monitors (SOM) as an alternative to programming with Java monitors. Monitors method calls are reified as sequential requests which are processed by a Scheduler. SOM follows the Actor [1] concept by which all messages to a SOM are processed sequentially. This potentially entails a loss of parallelism.

Since we want to avoid code pollution we did not consider using transactional approaches for solving the synchronization problem.

Svend Frolund and Gul Agha [5] introduce the notion of Multi-Object Coordination. In order to synchronize the behavior of threads they use patterns of coordination that can be defined abstractly. Patterns are expressed in the form of constraints that organize the invocation to a group of objects. Constraints are responsible for allowing access or not to a group of objects. Synchronizers are the abstractions that define the patterns to be used. They use the Actor approach.

One important drawback that we have perceived in all previous approaches is a lack of abstractions for modeling the synchronization requirements. Lopes provided a language construction for modeling synchronization specifications but it was not possible to extend or modify its predefined behavior.

7. CONCLUSIONS AND FUTURE WORK

We introduce synchronization specifications to separate of synchronization code from functional code. Since synchronization specifications are first-class objects, they are available at runtime to support dynamic adaptation to synchronization policies. Different locking mechanisms can be used to provide different levels of liveness and fairness. When new synchronization specifications are introduced, potential conflicts can be resolved.

We identify four main directions for further work:

- High level synchronization specifications.
- Synchronization policies.
- Coordinators and Schedulers
- Configuration

The synchronization specification is a high level abstraction which we would like to further develop in order to obtain more ways to express synchronization requirements. For instance, we would like to express that the use of a specific object by different threads should be fair. For some problems we should be able to simply state at a high level the situations that should be avoided (safety) or the activities which should always be enabled (liveness). We currently provide a mutually exclusive policy of objects' methods. We would like to support different kinds of policies that will control the adaptation process and will deliver a higher level of context sensitivity. For example we could have optimistic [8] and pessimistic policies. Depending on the circumstances of an application the policy could be switched between pessimistic and optimistic making the adapted code adapt to the context and change its behavior. Furthermore, in the absence of competing threads, thread safety is not an issue, so the context should be free not to impose any synchronization at all. We would like to analyze if the schedulers and coordinators abstractions presented in Section 6 are useful in this context. We also plan to extend the specification capabilities to detect potential problems and merge specifications that overlap.

Acknowledgements

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010), and CHOOSE, the Swiss Group for Object-Oriented Systems and Environments.

8. REFERENCES

[1] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[2] Denis Caromel, Luis Mateu, and Eric Tanter. Sequential object monitors. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004), number 3086 in Lecture Notes in Computer Science*, pages 316–340. Springer-Verlag, 2004.

[3] Marcus Denker. *Sub-method Structural and Behavioral Reflection.* PhD thesis, University of Bern, May 2008.

[4] Johan Fabry and Theo D'Hondt. Kala: Kernel aspect language for advanced transactions. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied*

computing, pages 1615–1620, New York, NY, USA, 2006. ACM.

[5] Svend Frolund and Gul Agha. A language framework for multi-object coordination. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 346–360, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[7] Joerg Kienzle and Rachid Guerraoui. AOP: Does it make sense? the case of concurrency and failures. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*. Springer Verlag, 2002.

[8] Hsiang-Tsung Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, June 1981.

[9] Cristina Isabel Videira Lopes. D: A language framework for distributed programming, 1997.

[10] Lukas Renggli, Stéphane Ducasse, and Adrian Kuhn. Magritte — a meta-driven approach to empower developers and end users. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 106–120. Springer, September 2007.