Making Architectural Analysis Reasonable

Andrew Berry
School of Computer Science
The University of Queensland
Australia 4072
andyb@dstc.edu.au

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA, 15213
garlan@cs.cmu.edu

November 2, 1998

1 Introduction

Software systems that integrate a set of concurrent and possibly distributed components are becoming increasingly common. One architectural style that is often used in such systems is implicit invocation[1, 2]. In this style, a component communicates and passes control by announcing events, and these events are multicast to a set of consuming components that perform actions in response to events. At first glance, it would seem that the inherent concurrency associated with this style would make systems intractable: the concurrency and independence of components coupled with event multicast typically leads to a highly complex concurrent system with considerable non-determinism.

We are currently exploring the problems of modeling and reasoning about highly concurrent systems at an architectural level. In particular, we have identified two approaches that help make such reasoning tractable:

- recognizing that an application matches a known pattern, thus allowing the use of simplified reasoning techniques;
- constraining the architectural style so that systems in that style can be more easily understood.

In this position statement, we focus on how the two approaches identified above can be applied to systems built using an implicit invocation style. We introduce a model of concurrent systems and show how the implicit invocation style produces a concurrent system. We then demonstrate how the approaches above can be applied using a number of examples. A formal treatment of this work is currently being developed. Our eventual goal is to develop a systematic understanding of the sources of tractability in concurrent systems and a set of techniques to assist developers in reasoning about such systems.

2 Reasoning about Concurrent Systems

A concurrent system can be modeled as a directed graph of actions. The nodes of the graph represent actions and the edges represent ordering dependencies between actions. Nodes not connected either directly or transitively are potentially concurrent. A system can be decomposed into a set of computations, which are subgraphs of the system with no edges connected to nodes outside the subgraph.

Concurrency in the system potentially leads to two types of conflict:

1. Concurrent access to shared state by distinct actions, for example, $x := x + 2$ and $x := x - 5$. The result of executing these two actions concurrently is undefined. These conflicts are typically called race conditions.

2. Modification of shared state by an independent action between dependent actions of a computation. For example, if a computation has two actions $x := x + 2$ followed by $x := x * 3$ and a postcondition that $x = (x + 2) * 3$, an independent action that executes $x := x - 5$ between the dependent actions will violate the postcondition. This is often called the isolation problem in concurrency literature.

The distinction between these two types of conflict can be somewhat fuzzy. In the second case, if we perform analysis at a level of abstraction that treats the dependent pair $x := x + 2$ and $x := x * 3$ as a single, indivisible action, then that action is concurrent with $x := x - 5$, and we have a race condition. This prevents us from developing a finer-grained concurrency control strategy in cases where some overlap is possible.

In order to show that a system behaves correctly in the presence of concurrency, we can take one of two approaches:

1. Prove that the system is correct when no conflicts occur, then show or ensure that conflicts are avoided. Since a system is typically defined as a set of distinct computations, we can prove that each computation is correct using existing development techniques, then show or ensure that their concurrent execution does not lead to any conflicts.
2. Show that the system will reach a correct state after its computations are complete, regardless of any possible conflicts. This approach is more difficult to decompose, and typically requires the recognition of patterns in the system.

Note that problems of deadlock, livelock and fairness are not addressed since we have not prescribed locking as a means to resolve conflicts.

3 Implicit Invocation

An implicit invocation system is defined by a set of methods, events, and a mapping between events and methods[1]. When an event is announced, all methods that have a mapping for that event are executed. This mapping is often realized by a dispatching component that maintains the event-method mapping. The concurrency and ordering of method executions is typically not specified, hence systems exhibit a high level of potential concurrency and non-determinism. Many systems allow the event-method mapping to be modified dynamically, for example through subscriptions by components, which increases the complexity.

In terms of the concurrency model described in the previous section, a method execution corresponds to an action, and an event announcement corresponds to an edge between action nodes or the initiation of a new computation if announced by an external influence. The dispatch mechanism defines templates for computation: each externally announced event has a computation graph defined by the mappings stored in the dispatcher.

In a system with unconstrained concurrency and asynchronous dispatching, the potential for conflicts of both types identified in the previous section is extremely high. For example, in an automated software development system where a compile action is fired whenever a particular file is written, writing the file during a compilation will potentially invalidate the compile and also fire a second compilation that will conflict with the original compile action. It is clear that techniques to either minimize conflicts or reason about their influence are necessary to reason about the correctness of an implicit invocation system.

4 Constraining Concurrency

The first and perhaps obvious approach is to constrain the implicit invocation style to reduce concurrency, and hence minimize conflicts. There are several common approaches used in practice.

4.1 Synchronous Dispatching

In a single-processor environment it is common to use a synchronous dispatch mechanism to realize the event-method mappings. There is no reduction in concurrency, since the processor can only executing one method at a time, regardless of the dispatch policy. Events are queued and dealt with in order of arrival. Each method associated with the event is called in a non-deterministic sequential order. The effect of this constraint is that race conditions between methods are avoided entirely and need not be considered when reasoning about the application semantics.

A further synchronization restriction is sometimes applied. It requires that each computation runs to completion before any other computations are started. The constraint imposes a total order over computations and removes any possibility of conflict between computations, meaning that an implicit invocation system can be proven correct by simply proving that each computation satisfies its postcondition and any global invariants. Note, however, that non-determinism in the ordering of method invocation might still be present within computations.

4.2 Serializing Conflicting Actions

In the presence of concurrency, it is possible to exclude race conditions by ensuring that conflicting methods do not execute concurrently. This requires that the dispatcher have knowledge of the resource access requirements of methods and be able to determine automatically if two methods conflict. This has the same effect as a simple synchronous dispatch mechanism. While race conditions are excluded, it is still necessary to prove that any conflicting interleaving of the actions in concurrent computations achieve a correct result.

4.3 Serializing Conflicting Computations

A dispatcher implementing serialization of entire computations ensures that all possibly conflicting computations are executed in a serializable order. As with method serialization, this constraint requires the dispatcher to have knowledge of the resource access requirements of methods and the ability to determine automatically if two computations conflict so it can schedule their execution to avoid conflicts. This has the same effect as executing all computations in a sequential order, while allowing concurrency where no conflicts exist. A common way of realizing serialization is through two-phase locking used in transaction and database systems, but this is not often provided for implicit invocation systems.

In a system applying serialization to entire computations, the system can be proven correct by simply proving that each individual computation satisfies the system semantics in isolation.

5 Application Patterns

An alternative approach to reasoning about highly concurrent executions in implicit invocation systems is to identify application patterns for which simplified reasoning techniques exist. This approach typically allows a higher level of concurrency since it can take advantage of semantic properties of the application. In this section we identify two patterns by presenting example problems and showing how the computations can be proven correct in the presence of concurrency.

5.1 Set/Counter Example

The set and counter example has two pieces of state: a set and a counter. There are two methods, *add* and *remove* that operate on the set and *inc* and *dec* methods that operate on the counter. The required behavior of the system is expressed as the invariant that, when all computations are complete, the counter reflects the number of elements in the set.

In an implicit invocation environment, the required semantics can be achieved by having the *add* and *remove* methods announce an event for each successful addition or removal of an element. The events for adding and removing elements are mapped to the *inc* and *dec* methods respectively. There are two key elements of this example that make reasoning more tractable:

1. The system semantics is expressed as a global invariant for the quiescent state. It is not necessary for the counter to be accurate after each modification to the set, only when the system is quiescent.
2. The semantics is realized entirely through the relationship between actions (method invocations) and events, *not* through access to intermediate state (i.e. the set). The interleaving of events and method invocations is therefore unimportant.

Note that the semantics also relies on the correct execution of each operation on the set or counter. This requires serialization of updates to the set or the counter to avoid race conditions. We can generalize this example and assert that if there are no interleaving conflicts and race conditions are avoided, application correctness can be shown by simply proving that each computation in isolation establishes the global invariant upon completion.

5.2 Edit/Compile Example

The edit and compile example has two pieces of state: a source file and an object file. There are two methods: *edit*, which modifies the source file, and *compile*, which generates an object file using the source file. The semantics

required of the system is expressed as a global invariant that, when the system is quiescent, the object file will be a correct compilation of the source file.

In an implicit invocation environment, the system semantics can be achieved by having each edit action announce an event on completion, and having an event mapping that causes a subsequent compile action. This works because the last action before a quiescent state will always be a compile action that is *not* concurrent with an edit action, and the invariant is established by the compile action. It is not necessary for edit actions to be serialized provided we assume that the writing of files is an atomic action and that there is no condition requiring exclusive access to the source file. It is necessary, however, for concurrent compile actions on a single source file to be serialized since they establish the invariant and must execute correctly. Again, there are two key elements of this example that make reasoning more tractable:

1. The system semantics is expressed as a global invariant for the quiescent state.
2. The invariant is established by the last action in any ordering of computations.

The pattern that emerges from this example is characterized by the two key elements captured above. If these conditions are satisfied, the correctness of computations can be shown by proving that the last action establishes the invariant.

The familiar model-view-controller pattern from user interface toolkits is a further example of this pattern. Such a system is correct provided all views reflect the state of the model component when the system is in a quiescent state. The last action(s) in any ordering are the view update actions, and these establish the invariant. It might be necessary, however, to serialize actions modifying the model component.

6 Discussion

Implicit invocation is an example of an architectural style that results in applications with a high level of concurrency. Reasoning about highly concurrent applications is difficult in the general case, and techniques are needed to deal with the concurrency at an architectural level. We have presented a two-pronged approach to this problem: applying generic constraints to the style that reduce concurrency and hence conflicts, and identifying application patterns that exhibit properties that make reasoning more tractable. We are currently developing a more formal treatment of the model and examples, with the goal of providing confidence in our results and demonstrating how the reasoning techniques can be applied in a more formal manner.

While our approach has focused on the implicit invocation style, we believe it could be equally applicable to other architectural styles. In particular, we believe that the model of concurrent systems and generic constraints presented here are usable in other styles with minor modification, but we expect that the application patterns emerging from other styles will be different. Partitioning the design space using architectural styles helps us to identify such application patterns and further assists in the architectural reasoning process. Our future work will focus on extending the results of this work to other architectural styles.

References

[1] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag, LNCS 551.

[2] K. Sullivan and D. Notkin. Reconciling environment integration and component independence. *ACM Transactions on Software Engineering and Methodology*, 1(3), July 1992.