4-2008

A scalable approach to multi-style architectural modeling and verification

Stephen WONG

Jing SUN

Ian WARREN

Jun SUN
Singapore Management University, junsun@smu.edu.sg

A Scalable Approach to Multi-Style Architectural Modeling and Verification

Stephen Wong, Jing Sun, Ian Warren
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland, New Zealand
swon149@ec.auckland.ac.nz
{j.sun, ian-w}@cs.auckland.ac.nz

Jun Sun
School of Computing
National University of Singapore
10 Kent Ridge Cresent, Singapore
sunj@comp.nus.edu.sg

Abstract

Software Architecture represents the high level description of a system in terms of components, external properties and communication. Despite its importance in the software engineering process, the lack of formal description and verification support limits the value of developing architectural models. Automated formal engineering methods can provide an effective means to precisely describe and rigorously verify intended structures and behaviors of software systems. In this paper, we present an approach to support the design and verification of software architectural models using the Alloy analyzer. Based on our earlier work, we propose a fundamental library for specifying system structures in terms of different architectural styles. We illustrate use of the architecture style library in modeling and verifying a complex system that utilizes multi-style structures. To promote scalability, we use model decomposition to parallelize the verification process. Results show that our approach enhances the performance of verifying models significantly.

1 Introduction

Software Architecture plays a key role in the software development process. Principally, an architectural model of a software system allows individuals to reason and communicate about the system at an abstract level, thereby hiding superfluous complexity. An architectural model is typically expressed in terms of components, properties and inter-component communication. The architectural modeling process is associated with various tools that support system modeling, analysis and verification, and which promote communication amongst different stakeholders [11].

Architectural Description Languages (ADLs) are an established means of expressing architectural models in terms of components and connectors [10]. Many researchers have investigated how to add formalism to ADLs in order to support formal verification. Notably, Allen and Garlan [2, 3] have described a CSP-like notation named Wright that can be used to formally express inter-component interaction. More recently, Garlan and Schmerl [6] have shown how formal definitions can be applied to ADLs to create a formally-based architectural centric approach to software design. Such approaches aim to improve software quality by offering a way of formally verifying software at an early stage in the development process.

The aim of verification is to determine whether a modeled system will satisfy given properties. Formal verification takes a mathematical approach and relies on a formally described system model. Formal description could help express properties such as interaction, load balancing, availability and security in a rigorous manner [4]. Automated verification potentially offers an efficient means of verifying a system model. However, one drawback of many existing approaches to automated formal verification is their limited scalability, where large size models are computationally expensive to process. A key aspect of the work described in this paper is to address the scalability problem.

Following early work [1] in modeling system architectures using components and connectors, there has been much interest in specifying architectural styles. A style essentially imposes constraints on types of components, connectors and their assembly into a system. For example, the pipe and filter style [13] dictates that components be stateless and that components be connected to form a linear chain, where components accept data on their inbound connection, transform the data, and output the transformed data on their outbound connection. Recent work done by Kim and Garlan [9] also proposed modeling and verification of architecture styles using the Alloy formal language. In their approach, a few architectures were modeled based on ACME [12] ADL definitions. It offers a useful insight to the ability of using Alloy to verify properties within/between architecture styles such as inter-operability. However, the

978-0-7695-3139-7/08 $25.00 © 2008 IEEE
DOI 10.1109/ICECCS.2008.16

scalability issue remains as a practical limitation of the research. The problem arises from the large scope of problems expanding the search space for verification of the Alloy reasoning tool.

Large and complex software systems are often best modeled using a combination of architectural styles. In this paper, we propose an approach that allows systems to be modeled and composed by a formally defined reusable library of fundamental architectural styles. Using the library, a system model can be easily and rapidly constructed. The resulting model captures the intended structures and properties that each specific style structure must hold, which is typically augmented with system-specific properties and constraints. Furthermore, in order to promote scalability of the verification, we apply a model splitting approach to identify subsystems that conform to particular architectural styles and perform parallelized verification accordingly. In essence, a subsystem becomes a unit of formal verification that can be verified in parallel to other units.

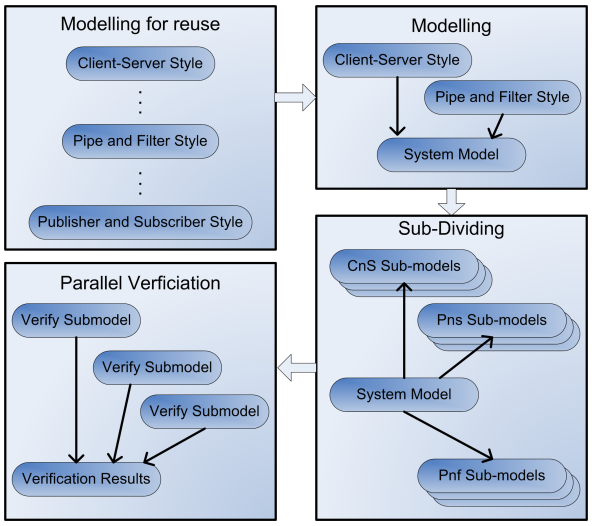

Figure 1. The Overall Modeling Approach

Figure 1 presents an overview of our approach. The first activity, '*Modeling for reuse*', involves defining the set of reusable architectural styles. In practice, this step also includes the verification of these pre-defined architecture styles, since each style has its associated behaviors and corresponding properties. These verified styles can be used to construct more complicated architecture structures for modeling target systems. In the second step, '*Modeling*', a system model is constructed by drawing on styles held in the library. Library styles can be extended and augmented as necessary to model the target system. In preparation for formal verification, subsystems that conform to particular architectural styles are automatically detected in the '*Sub-Dividing*' phase for distributed verification. The final activity, '*Parallel Verification*', employs distributed instances of the Alloy tool to formally verify each subsystem in parallel and report the results.

The remainder of this paper is structured as follows. In Section 2, we present an overview of the Alloy notation and its associated tool. In Section 3, we present formal modeling of the architectural style library that includes a number of popular architectural styles. Section 4 illustrates use of the architecture style library in modeling and verifying a multi-styled system as a case study. In Section 5, we describe the model splitting approach where a system model is decomposed into subsystems for distributed verification. Prior to drawing conclusions and outlining the future work, in Section 6 we present results based on an initial scalability evaluation.

2 Alloy

Alloy [8] is a structural modeling language based on set theory and first order logic. It is suitable for expressing complex structural constraints and behaviors of software and hardware systems. The main language constructs of Alloy include: signatures, facts, functions, predicates and assertions [7].

- Signatures (sig) - Describe a particular set of object entities. These descriptions can contain relationships with other signatures as well as constraints on those relationships. A signature may extend another signature such that all relationships and constraints for the extended signature hold true for the new signature.
- Facts (fact) - Statements describe a particular set of constraints that are invariably true within a model. These facts can utilize any signatures, functions or predicates available.
- Functions (fun) - Functions are expressions that can accept parameters to evaluate relations. A function will return a value, and can be invoked providing any necessary parameter values.
- Predicates (pred) - Predicates are constraints that can accept parameters. They evaluate to either true or false, and can be invoked providing given parameter values.
- Assertions (assert) - Assertions are theorems that are required to be proved. They can be used to check if a model has certain properties and whether it satisfies various constraints.

Alloy Analyzer allows the models written in Alloy to be verified automatically in two ways, i.e., simulation and checking. Both techniques require a certain scope of instances to be specified. Simulation is used to ensure that

systems are feasible and that a model has not been over constrained with conflicting constraints.

Checking is done through searching the problem space within the given scope to identify any possible solutions that would violate its constraints. After a solution has been found, the Alloy Analyzer visualizes it in different formats, such as graph, dot, XML or tree representations. The latest version of Alloy Analyzer 4 incorporates a new constraint solver called KodKod [14], which improves efficient searching of a problem space by introducing partial instances.

3 Multi-Style Architecture Library

As we mentioned earlier, complex system structures can be modeled through a composition of different architecture styles. In this section, we present a multi-style architecture library in Alloy to support a reusable and extensible modeling approach for this purpose. Furthermore, the specified architecture style library also provides a good basis for distinguishing key sub-models in the later parallel verification stage.

3.1 The component and connector model

Architecture modeling focuses on the high level description of a system in term of the elements in the system and the interactions among the elements [5]. Components and connectors are two fundamental units in an architecture description. Components describe the identifiable computation entities of a system, where connectors specify the patterns of communication among these entities.

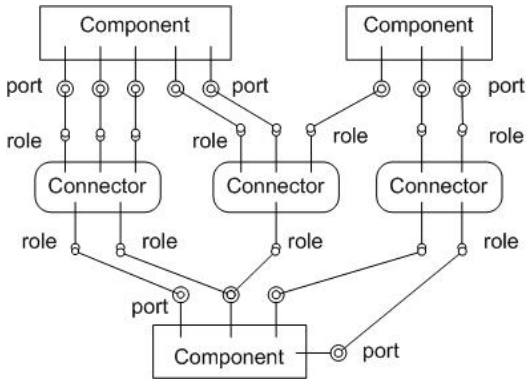

Figure 2. An example view of the component and connector style.

Figure 2 illustrates a simple example of such a structure. In the diagram, each component consists of a set of *port*s as its external visible interfaces. These ports act as communication points between the component and other components or the environment. For example, a port can be a requested interface that obtains information from others, or a provided interface that outputs information.

A connector consists of a set of *role*s that are used to describe the pattern of a particular kind of communication. For example, the connector in a Pipe-Filter style describes a directional and stream communication pattern that reads data from an input role and passes it on through an output role. In order to establish a connection between different components of the system, the roles of a connector must be attached to the ports of the participating components. The use of connectors brings the following three advantages to architecture modeling.

- It separates the concern of computation from communication in a system description, where components only focus on computation and connectors take care of the communication.
- It further promotes encapsulation and reusability in architecture modeling. Components and connectors with same/similar computation functionalities and communication patterns can be reused/extended easily.
- It provides a more flexible means of modeling different interactions among components. As each connector describes a specific pattern of communication, a component can participate in various types of interactions in the system through different connectors. For example, the output port of a component can be attached to a role in a Client-Server connector to send out requests to a server, as well as it can be attached to a role in a Pipe-Filter connector to send out streamed data commands to a log file, such as shown in Figure 2.

The following gives an Alloy specification of the component and connector style.

3.1.1 Components

```
abstract sig Component {
  ports : set Port,
  action : set Process -> lone Process
}{
  this = ports.component
  action.Process.parent in ports
  action[Process].parent in ports
}
abstract sig Port{
  component : one Component,
  portProcesses : set Process
}{
  this in component.ports
  this in portProcesses.parent
}
abstract sig Process{
  parent : (Port + Role)
}{
  this in parent.(portProcesses + roleProcesses)
}
```

The above defines three signatures, i.e., `Component`, `Port` and `Process`. A port specifies the component it belongs to and a set of processes that it is entitled to perform. A process represents a generalized means of a behavior description, e.g., a 'write' operation for the port of a filter component in the Pipe-Filter architecture style. A component consists of a set of `ports` and its `actions`. The `actions` is a relation between processes that captures inter-behavior sequences of a component. For example, the action of a client component in a Client-Server structure can perform sending a request and then receiving a result. A process can be associated with a port or a role in describing their behavior patterns.

3.1.2 Connectors

```
abstract sig Connector {
  roles : set Role,
  dataflow : set Process -> set Process
}{
  this = roles.connector
  dataflow.Process.parent in roles
  dataflow[Process].parent in roles
  irreflexive[dataflow]
}
abstract sig Role{
  connector : one Connector,
  roleProcesses : set Process
}{
  this in connector.roles
  this in roleProcesses.parent
}
```

A `role` specifies the `connector` it belongs to together with a set of processes that it is entitled to perform. A `connector` consists of a set of `roles` and its `dataflow`. The `dataflow` is a relation of processes for modeling the behavior patterns of a communication, it is set as a relations to allow mutliple broadcasting connector type such as eventbus where one event may be received by multiple members. For example, the dataflow of a connector in a Client-Server structure can perform either receiving a request and invoking a corresponding service or obtaining a server response and returning it to the client. In addition, a `dataflow` is made irreflexive to disallow behavior relationship between the same processes.

3.1.3 Attachments of a system

```
abstract sig System{
  components : set Component,
  connectors : set Connector,
  attachments : Role -> lone Port
}{
  attachments.Port in connectors.roles
  attachments[Role] in components.ports
}
fact SameProcess{
  all p :Port , r : Role |
    r->p in System.attachments =>
      r.roleProcesses in p.portProcesses
}
fact noLooseObject{
  all c:Component, con : Connector |
    some s : System | c in s.components
      && con in s.connectors
}
```

The above shows a structural view of the component and connector style, where a system contains a set of `components`, a set of `connectors` and the `attachments` between them. The `attachments` are defined between roles and ports, where a port can be attached to different roles, but not verse versa. This provides flexible descriptions which allow the interface of a component to play different roles in various communications. The constraints part of the `System` ensures that the participating components can communicate with the desired connector and are expected to play the given roles.

Two fact specifications further ensure a correct structure of the style. The fact '`SameProcess`' states that for each attachment established between a port and a role, the processes of the role are a subset of the processes of the port, which ensures that the attached port can offer all the operations required from the role. The fact '`noLooseObject`' states that all components and connectors should be attached in the style, which prevents the existence of isolated components or connectors in a given model. Furthermore, for capturing the state changes of a system, the following defines two parameterized predicates for attaching/detaching a port with a role.

```
pred attached(s:System,s':System,r:Role,p:Port){
  s'.attachments = s.attachments ++ r->p
}
pred detached(s:System,s':System,r:Role,p:Port){
  s'.attachments = s.attachments - r->p
}
```

Based on the above definitions, we can run the Alloy Analyzer to create a snapshot of the component and connector style similar to that of shown in Figure 2.

3.2 The Client Server Model

One of the most commonly used architecture styles is the client server structure. It offers a loosely coupled multi-connection mechanism in a consumer and provider fashion. A server can be viewed as a component that provides a set of services. This set of services are exposed to the various clients for consumption. The external behaviors of the server vary in specific systems, however, two generalized operations can be observed as receiving and responding to client requests. A client can be viewed as a consumer component of the server. It mainly performs two types of operations that request and obtain results from different services.

Furthermore, the client and server structure also has a set of properties (protocol or contract) for its service consumption. For instance, for each valid service request from a client, there should be a corresponding response from the server to the client within a certain response period.

3.2.1 Server and clients

The following gives an Alloy specification of the client and a server style.

```
abstract sig Server extends Component{} {
  some ports & ClientAccess
}
abstract sig Client extends Component{} {
  some ServerRequest & ports
}
abstract sig ClientAccess extends Port{} {
  some component & Server
  some Invoke & portProcesses
  some Return & portProcesses
}
abstract sig ServerRequest extends Port{} {
  some Request & portProcesses
  some Result & portProcesses
}
```

The above defines four signatures, `Server`, `Client`, `ClientAccess` and `ServerRequest`. A `ClientAccess` extends the definition of a `Port` and has two associated processes, `Invoke` and `Return`. A `ServerRequest` is a special kind of `Port` that can perform the `Request` and `Result` processes. A `Server` extends a component signature and has `ClientAccess` as its ports; while a `Client` is a special type of component that has `ServerAccess` as its ports.

Furthermore, properties that are specific to a Client-Server style can also be modeled accordingly. For example, the following fact '`InvokeLeadstoReturn`' states the property that if a request is received at the server from a client invocation, a corresponding result shall be returned as its consequence.

```
fact InvokeLeadstoReturn{
  all s: Server| some i:Invoke, r:Return|
    some s.action.Process & i
      => some s.action[Process] & r
}
```

3.2.2 Connection

```
abstract sig CnsConnector extends Connector{}{
  some Provider & roles
  some Requester & roles
  Request -> Invoke in dataflow
  Return -> Result in dataflow
}
abstract sig Provider extends Role{} {
  some connector & CnsConnector
  some Invoke & roleProcesses
  some Return & roleProcesses
}
abstract sig Requester extends Role{} {
  some connector & CnsConnector
  some Request & roleProcesses
  some Result & roleProcesses
}
fact SomeRole{
  all c : CnsConnector|
    some p : Provider, r:Requester |
      p in c.roles <=> r in c.roles
}
```

There are only two types of roles in a client and server connection, i.e., the `Provider` and the `Requester`. A `Provider` can perform the operations such as `Invoke` and `Return`; while the `Requester` plays the role of `Request` and obtains a `Result`. A connector in a client server structure, `CnsConnector`, has a set of providers and requesters as its roles. And its behavior descriptions are modeled by two types of data flows, i.e., '`Request -> Invoke`' (client requests and server invokes) and '`Return -> Result`' (server returns and client receives). Finally, the fact '`SomeRole`' ensures that both the provider and requester must be present in order to establish a client-server connection.

Based on the above specification, we can run the Alloy Analyzer to create a snapshot of the client and server style. Figure 3 shows running instances of a server with two clients, where each client establishes a separate connection to the server, accessing the same interface (port) that the server provides.

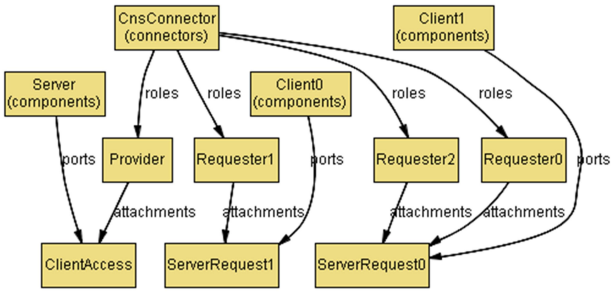

Figure 3. Alloy instances of the Client-Server style

In this section, we demonstrated the Alloy modeling of two commonly used architecture styles, i.e., Component-Connector and Client-Server. To construct a multi-style architecture library for reuse, we have also defined other styles, such as Publisher-Subscriber, Pipe-Filter and so on. Due to the page limitation, we will not present their specifications here. The completed Alloy style library has been made available at `http://www.cs.auckland.ac.nz/~jingsun/multi-style/library/`.

4 Case study - A Web Site System Example

In this section we present the modeling and verification of a web site system example that illustrates use of the multi-style architecture library defined in the previous section.

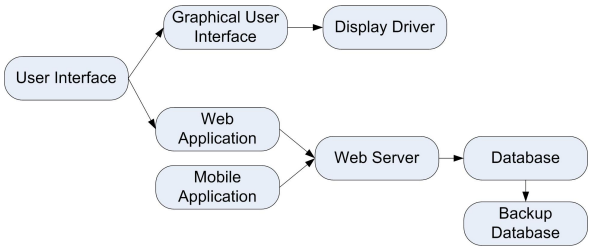

Figure 4. A simple web site system overview

A simple web site system describes the interactions from the user input to the web server and its database, as shown in Figure 4. It consists of components such as user interface, display manager, application, server and database. Based on multi-style modeling, this system can be identified to utilize three different architecture structures, i.e., Client-Server , Pipe-Filter and Publish-Subscriber, which can be further specified as follows.

4.1 User interface, display driver and GUI

User interface can be viewed as a component that handles various inputs from the users and assigns commands to other parts of the system. Thus it can be modeled as a publisher in the Publisher-Subscriber style as follows.

```
sig UserInterface extends Publisher{}{
  some DataEvent & ports
  some HumanInput & ports
  dataInput -> PublishData in action
}
```

The `DisplayDriver` component receives data from the `UserInterface` component and writes it to a buffer for the Graphical User Interface (GUI) which forwards data to the display driver. It can be considered as a filter component in the Pipe-Filter style, which can be modeled as follows.

```
one sig GraphicalUserInterface extends Filter{}{
  DisplaySubscriber + SendDisplayChanges = ports
  ReceiveDisplayData->WriteDisplayBuffer = action
}
one sig DisplayDriver extends DataSink{}{
  ReceiveDisplayChanges = ports
  no action
}
sig DisplayChannel extends Pipe{}{
  WriteDisplayBuffer -> ReadDisplayBuffer +
    EOF -> CloseConnection = dataflow
}
```

A `GraphicalUserInterface` component contains `DisplaySubscriber` and `SendDisplayChanges` as its interfaces. These two ports describes the input and output of the GUI respectively. The second component is the `DisplayDriver` being a data sink. The `DisplayChannel` shows that the dataflow is from a writer to a reader, however an `EOF` (end of file) indicates that the connection will be closed by the Reader.

4.2 Application, web server and database

The application and server structure reassembles the Client-Server style. A client application is responsible for retrieving requests from the `UserInterface` and sending requests to the server enquiring about the information needed. The server then accesses the database to query specific information to return it to the client. There are two types of clients available in the system, i.e., web application and mobile application, which can be modeled as follows.

```
abstract sig Application extends Client{}{
  some ServiceInvocation & ports
}
sig WebApplication extends Client{}{
  some DataSubscriber & ports
  some WebServiceInvocation & ports
  ReceiveData -> SearchRequest = action
}
sig MobileApplication extends Client{}{
  some MobileServiceInvocation & ports
}
```

A web server links between the applications and the database, hiding the business logic. Its connection (TCP/IP) extends a `CnsConnector` and consists of two types, i.e., broadband and wireless.

```
abstract sig WebServer extends Server{}{
  some RequestAccess & ports
  some DatabaseQuery & ports
  InvokeSearch -> RequestDBQuery +
  DBQueryResult -> ReturnSearchData = action
}
abstract sig TCPIPConnection
                        extends CnsConnector{}{
  SearchRequest -> InvokeSearch +
    ReturnSearchData -> SearchResult in dataflow
}
sig BroadBand extends TCPIPConnection{}
sig WirelessConnection extends TCPIPConnection{}
```

The database can be considered as a server for the web server, holding information that the server would request. It was modeled by a replication technique, creating a main and a backup database. Each database is capable of accepting queries and returning the information required. The main database is also capable of sending information to its replica. A database connection is defined as an extension of the `CnsConnector` as follows.

```
abstract sig Database extends Server{}{
  some DBAccess & ports
}
one sig MainDataBase extends Database{}{
  some DataChange & ports
  some DBAccessMain & ports
  InvokeDBQuery -> ReturnData +
  InvokeDBQuery -> DataWrite = action
}
one sig BackupDatabase extends Database{}{
  some DataUpdate & ports
  some DBAccessBack & ports
  InvokeDBQuery -> ReturnData = action
}
...
abstract sig DataBaseConnection
                    extends CnsConnector{}{
  RequestDBQuery->InvokeDBQuery +
    ReturnData-> DBQueryResult = dataflow
}
```

After defining the basic elements of the web site system, we can specify a structure predicate to describe state changes on the dynamic aspects of the system as follows.

```
pred Structure(s:WebSys, s':WebSys){
  PublishAnEvent[s,s',UserInterface,DataEventBus,
    DataEvent] && ...
  SubscribeToEvent[s,s',GraphicalUserInterface,
    DisplayEventBus,DisplaySubscriber]&& ...
  ClientConnection[s,s',MobileApplication,
    WirelessConnection,MobileServiceInvocation]
  && ...
}
```

The above predicate constraints the model to give a proper and intended structure, e.g., `PublishAnEvent` ensures the connection to an event bus of a publisher or subscriber, etc. Based on the specification, we can run the Alloy Analyzer to create a snapshot of the system [1].

4.3 Checking system properties

Apart from generating snapshots of the running system, we are also able to perform assertion checks on desired properties that must hold in the system using the Alloy Analyzer. Note that some of these properties are specific to the architecture styles that used in the system and others are related to the the system itself. For example, the following assertion specifies that any updates on a publisher will result in notifications to all its subscribers.

```
assert CorrectFlow{
 all s:System, conn:EventBus,pub:Publisher,
   sub:Subscriber|some pp:pub.ports,sp:sub.ports,
     a:pp.portProcesses, l:sp.portProcesses|
      pp in s.attachments[conn.roles] &&
      sp in s.attachments[conn.roles] &&
      some a & Announce && some l & Listen =>
       a -> l in conn.dataflow
}
```

The assertion `CorrectFlow` defines if a `Publisher` and a `Subscriber` share an `EventBus` connection, there must be a flow from the `Announce` process to the `Listen` process respectively. If we run the Alloy Analyzer to check the above assertion, it returns '*No counter-example found*', which indicates that no counter-examples can be generated, thus the assertion is true.

The second example ensures a fault tolerance property among the web server and its two databases.

```
assert backup{
all s:WebSys | Structure[s,s]  &&
 some conn : DataBaseConnection |
  (no s.attachments.DBAccessMain in conn.roles &&
   some s.attachments.DataQuery in conn.roles) =>
   (some s.attachments.DBAccessBack in conn.roles
    && some s.attachments.DataQuery in conn.roles)
 && some p:Pipe |
  some s.attachments.DBAccessMain in p.roles
   && some s.attachments.DBAccessBack in p.roles
}
```

The above assertion states two properties, i.e., (1) if the connection (`DBAccessMain`) between the web server and the main database is lost, another connection (`DBAccessBack`) from the web server will automatically switch to the backup database; (2) there should always be a connection between the main and backup database. This assertion can be checked via the Alloy Analyzer.

In the following example, a load balancing property is specified, where the number of connections between all server's attachments should only differ by at most one.

```
assert LoadBalanced{
 all  s:WebSys, s':WebSys-s |
   |all ws:WebServer, ws':WebServer-ws |
     Structure[s,s'] &&
       #s.attachments.(ws.ports) =
         #s.attachments.(ws'.ports) ||
       #s.attachments.(ws.ports)+1 =
         #s.attachments.(ws'.ports) ||
       #s.attachments.(ws.ports)-1 =
         #s.attachments.(ws'.ports)
}
```

In this section, we demonstrated the modeling of a simple web site system based on the reuse and extension of different architecture styles defined in section 3. We used Alloy Analyzer to apply automated verification on desired system properties. However, the performance of such a verification does not show a practical solution to the problem. For instance, the running time of a scope of 18 Processes and 9 Components on the web site system model took 307924 milliseconds (5.2 minutes) to execute. Such a performance could not meet any practical verification needs since a more

[1]For the complete specification of the web site system example, please refer to `http://www.cs.auckland.ac.nz/~jingsun/multi-style/websitesystem.als` for modeling and verification details.

typical computer system may consist of hundreds of components (instances). To overcome this disadvantage, we further propose a model splitting approach to accommodate distributed parallel verification.

5 Model Splitting and Parallel Verification

The model splitting approach is a methodology developed to improve the efficiency of the verification processes. It aims at decomposing a complex architecture model into various smaller sub-models to allow parallel verification. Scalability of the verification can be greatly improved by decreasing the size of the model (scope of the Alloy instances). This section details the necessary steps in dividing a model for parallel computation according to its architecture structures. The previously mentioned web site system example of Section 4 is used to illustrate the model splitting process.

5.1 Identifying sub-models

The key issue in a modeling splitting approach is to identify sub-models for decomposition. As we mentioned earlier, such a decomposition could be based on the architecture structure of a system. Each identifiable architecture style in the model provides a perfect grouping of a subsystem that performs specialized tasks. A sub-system is a smaller model that contains less scope, hence, easier to verify. Together all sub-systems can be verified in a distributed manner and the results of the verification can be collected in a more efficient way. In general, our basic approach in model splitting is similar to the modular programming technique as follows.

- Firstly, identify the number of architecture styles available in the system. For each style instance, a submodel is created respectively.
- Secondly, for each global constraint (i.e., fact, predicate, function, assertion) defined in the model, localize it to become the constraints inside the corresponding sub-models. This process may involve the re-definition of these constraints in terms of the components and connectors inside the sub-models, as well as the creation of additional constraints among the identified sub-models.
- Finally, a simplified system model may be introduced to handle the high level constraint definitions among the sub-systems. This high level model can also be used to collect the running results of parallel verification.

In addition, hierarchial structure can be established among the sub-models, such as the high level system description consists of identified sub-style models, and each sub-model may further contain other sub-systems inside. For instance, a composite Client-Server style may contain a Pipe-Filter structure as one of its sub-systems. Thus submodel can be viewed as a composite component which may also have a set of interfaces (ports) as its external representation. Those interfaces are used as the interaction points regarding communication among the sub-systems and the rest of the system.

The above presents a brief guideline for model decomposition. It can offer a systematic approach to divide and deliver multiple sub-models for parallel verification. The following is a pseudo algorithm detailing the various steps it takes to sub-divide components into sub-models.

```
SubDivide(components, connectors, attachments) {
 let sm be the current submodel;
 sort the connectors from loose to tight coupling;
 take the loosest type from connectors as conn;
 let connset be the set of connectors in sm;
 let cs be the set of components in sm;
 for each c attached to conn
  if (c is a publisher){
    cs += c;
    connset += all EventBus connected to c;
    cs += all subscriber connected to
          EventBus in cs;
  }
  else if (c is server){
    ...
  }
  else if (c is filter){
    ...
  }
  for all pc in cs {
   if pc is connected to (connectors - connset) {
     generalize pc as a sub-system with
       name 'pc-subsystem';
     replace pc as 'pc-subsystem' in cs;
     let pcs be the set of all components
         reachable from pc outside of connset;
     let pconnset be the set of all connectors
         reachable connectors outside of connset;
     SubDivide(pcs, pconnset, attachments);
   }
  }
  include cs and connset in sm;
  add sm to the set of recognized submodels;
}
```

The algorithm performs the model splitting on a system model via recursion using the set of components, connectors and their attachments as inputs. Firstly, all the connectors are sorted based on their coupling aspects, i.e., in the order of Publish-Subscriber, Client-Server and Pipe-Filter. This represents the precedence of different types of connections in the system, which keeps the tightly coupled connections to form sub-systems first, then use them to form other sub-systems with the less coupled connections. Sec-

ondly, after the loosest connector is selected, the algorithm extracts all the attached components and connectors in relation with the loosest connector and uses them to form a sub-system with respect to different styles. Thirdly, for each component in the created sub-system above, if the component is also attached with other connectors outside this subsystem, a recursive call is made on all its reachable components and connectors together with the attachments to form a new sub-system. And this new sub-system is used to replace the component in the current sub-model. Once all the recursive calls returned, every component and connector in the original model will be divided to their corresponding sub-systems.

Applying the above component splitting algorithm to the example web site system in section 4, we could generate 5 sub-models accordingly. The following shows part of the first extracted Publisher-Subscriber sub-model, where two other sub-systems `DataSubSystem` and `DisplaySubSystem` are recursively identified. Please note that the original `GraphicalUserInterface` and `WebApplication` components that was connected to the component `UserInterface`, is now encapsulated and replaced by the two sub-systems `DataSubSystem` and `DisplaySubSystem`.

```
sig UserInterface extends Publisher{}{......}
sig GUISubSystem extends DataSink{}{
  some DisplaySubscriber & ports
}
sig WebApplicationSubSystem extends Subscriber{}
{
  some DataSubscriber & ports
}
......
```

Recursion of the algorithm can generate the remaining of 4 sub-models accordingly, i.e., a Pipe-Filter system over the `GUI` and `DisplayDriver` units, two Client-Server systems over the `Application`, `Server` and `Database`, and a final Pipe-Filter system over the `MainDatabase` and `BackupDatabase`.

After identified components into their corresponding sub-models, the next step is to localize the global constraints. In general, a global constraint is defined in terms of the relationships among the interfaces of the participation components. If we consider each sub-model as a composite component that has a set of external visible interfaces for communicating with other sub-models, then the following can be applied to the constraint splitting case: (1) if the constraint only involves components from the same sub-model, simply put the constraint into the sub-model; (2) if the constraint involves components from two or more different sub-models, re-write the constraint in terms of the interfaces from the sub-models. For example, let us consider the following assertion `CheckDataFlow` in the original model, which states that a data request from the user would lead to updates on the display as well as the database being queried.

```
  assert CheckDataFlow{
  SearchRequest in Component.action.dom =>
    InvokeDBQuery in Replies[SearchRequest,
      Component.action+Connector.dataflow]
}
```

By inspecting the variables (interfaces) involved the assertion, we can easily identify which sub-models they belong to. That is, `SearchRequest` belongs to the `WebApplicationSubSystem` and `InvokeDBQuery` belongs to the `WebServerSubSystem`. Please note that the `WebServerSubSystem` is a sub-model inside the `WebApplicationSubSystem`. After realizing the sub-models involved, the next step is to identify the common interfaces of the sub-models that enabled the communications. In this case, the interface `RequestAccess` contains a process `InvokeSearch` that represents the inputs into the `WebServerSubSystem`. With the linking process `InvokeSearch` being identified, an assertion for the `WebApplicationSubSystem` could be generated by replacing the `InvokeDBQuery` as follows.

```
module WebApplicationSubSystem
  ...
  assert CheckDataFlow{
    SearchRequest in Component.action.dom =>
      InvokeSearch in Replies[SearchRequest,
        Component.action+Connector.dataflow]
}
```

Similarly, another assertion can be introduced into the `WebServerSubSystem` through the common interface `InvokeSearch` as follows.

```
module WebServerSubSystem
  ...
  assert CheckDataFlow{
    InvokeSearch in Component.action.dom =>
      InvokeDBQuery in Replies[InvokeSearch,
        Component.action+Connector.dataflow]
}
```

This assertion is local to the `WebServerSubSystem`, which represents the second half of the original assertion that ensures if inputs happen through the `InvokeSearch`, a database invocation shall follow. The above gives a brief illustration about the constraint splitting. An algorithm for splitting the constraints into sub-models can be developed similarly to automate the sub-dividing process.

5.2 Verification and Evaluation

After the sub-models have been created from a complicated system model, we can perform parallel verification to run the sub-models in a distributed manner to improve the

efficiency of verification. The results from the divided Alloy models show promise. Due to the fact that each smaller model requires a smaller scope to be verified, it dramatically decreased the total time in which to verify the model as a whole. Some evaluation results are presented as follows. This evaluation was carried out using an Intel (R) Core (TM) 2 CPU 6400 @ 2.13GHZ, 2.13GHZ with 2.00 GB Ram computer. The time taken to execute the models, with their respective scope is presented in Table 1.

Model	Scope	Time(ms)
Full Model	18	307924
Pns Sub Model	4	78
Cns Sub Model1	8	172
Cns Sub Model2	5	188
Pnf Sub Model	5	62
Pnf Sub Model2	6	93

Table 1. Time taken to process a large system and its sub-models

Sub-models after splitting might be executed in a sequential manner due to their dependencies. In this case, "Pnf2 → Cns2 → Cns1 → Pns". The dependencies is caused by the subsystems within a model, e.g., 'Pnf2' is a subsystem within 'Cns2', therefor it needs to be processed first. This chain reaction would require roughly 531 milliseconds (=78+172+188+93ms). It is still a significant improvement than that of the execution time of the full sized model, which is about 307924 ms. These statistics support the theory of our model splitting approach as well as the fact that it is an effective method that allows a multi-style architecture modeling and verification to be feasible and practical in a scalable manner.

6 Conclusion

In this paper we presented a multi-style approach to software architecture modeling and verification. A library of styles is defined using the Alloy formal language to assist the reuse and extensible modeling of complex computer systems. We demonstrate use of the style library in the modeling of a web site system. It shows the capability of offering a clear method of expressing software architecture in a formal manner. It also provides a platform for which different properties of software architecture could be expressed as well as automatically verified via the Alloy Analyzer. Furthermore, to improve the performance of verification, a model splitting and parallel verification approach was proposed. By creating various independently verifiable submodels, this approach has enabled verification to be done in parallel, which increases dramatically the scalability of Alloy verification.

In future, we plan to further automate the model splitting and verification process. This includes the automatic generation of sub-models from a system description, and distributing the Alloy sub-models using distributed computation middleware allowing an effective way to verify system models in close to real time fashion.

References

[1] G. Abowd, R. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Trans. Softw. Eng. Methodol.*, 4(4):319–364, 1995.

[2] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[3] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[4] C. Chen, P. Grisham, S. Khurshid, and D. Perry. Design and Validation of a General Security Model with the Alloy Analyzer. First Alloy Workshop'06, available at `http://alloy.mit.edu/workshop/papers/chen.pdf`, 2006.

[5] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documentaing Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[6] D. Garlan and B. Schmerl. Architecture-driven Modelling and Analysis. In *Proceedings of the 11th Australian workshop on safety critical systems and software*, pages 3–17, Darlinghurst, Australia, 2006. Australian Computer Society.

[7] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[8] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[9] J. S. Kim and D. Garlan. Analyzing Architectural Styles with Alloy. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70–80, New York, NY, USA, 2006. ACM.

[10] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the 6th European Software Engineering Conference*, pages 60–76, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[11] H. Miao, J. Sun, and X. Cao. Formalizing and Analyzing Service Oriented Software Architecture Style. *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, pages 387–390, October 2006.

[12] R. Roshandel, B. Schmerl, N. Medvidovic, D. Garlan, and D. Zhang. Understanding Tradeoffs among Different Architectural Modeling Approaches. *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture*, pages 47–56, June 2004.

[13] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[14] E. Torlak and G. Dennis. KodKod for Alloy Users. First Alloy Workshop'06, available at `http://alloy.mit.edu/workshop/papers/torlak.pdf`, 2006.