Verification of Component-Based Software Application Families*

Fei Xie[1] and James C. Browne[2]

[1] Dept. of Computer Science, Portland State Univ., Portland, OR 97207
xie@cs.pdx.edu
[2] Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712
browne@cs.utexas.edu

Abstract. We present a novel approach which facilitates formal verification of component-based software application families using model checking. This approach enables effective compositional reasoning by facilitating formulation of component properties and their environment assumptions. This approach integrates bottom-up component verification and top-down system verification based on the concept of application family architectures (AFA). The core elements of an AFA are architectural styles and reusable components. Reusable components of a family are defined in the context of its architectural styles and their correctness properties are verified in bottom-up component compositions. Top-down system verification utilizes architectural styles to guide decomposition of properties of a system into properties of its components and formulation of assumptions for the component properties. The component properties are reused if already verified; otherwise, they are verified top-down recursively. Architectural style guided property decomposition facilitates reuse of verified component properties. Preliminary case studies have shown that our approach achieves order-of-magnitude reduction on verification complexities and realizes major verification reuse.

1 Introduction

Model checking [1] has great potential in formal verification of software systems. The massive effort required for model checking whole systems "from scratch" has, however, hindered application of model checking to software. The observations that many software systems are members of families of related systems which share common architectural styles and common components and that compositional reasoning [2, 3] is one of the most effective methods for reducing model checking complexities suggest component-based software verification, where verification of whole systems is based on compositional reasoning and on reuse of verified component properties.

A key challenge in component-based verification is formulation of component properties and their environment assumptions, i.e., what properties to verify on a component and what are the assumptions under which the properties should be verified. This challenge is largely due to lack of knowledge about possible environments of components. In the state of the art, property and assumption formulation is often ad-hoc and system-specific. There has been recent research [4, 5] on automatic generation of assumptions

* This research was partially supported by NSF grants IIS-0438967 and CNS-0509354.

I. Gorton et al. (Eds.): CBSE 2006, LNCS 4063, pp. 50–66, 2006.

for safety properties of components. However, formulation of component properties and formulation of assumptions for liveness properties still needs to be addressed.

This paper presents and illustrates a novel approach which facilitates formal verification of component-based software application families using model checking. This approach contributes to addressing the component property and assumption formulation challenge through extending the concept of software architectures to the concept of application family architectures (AFA). An AFA of an application family consists of the computation model, component model, architectural styles, and reusable components of the family. Intuitively, the AFA concept addresses lack of knowledge about possible environments of components by capturing common usage patterns and compositions of components and provides a hierarchy of reusable components with verified properties.

In this approach, *bottom-up component verification* and *top-down system verification* are integrated based on assume-guarantee compositional reasoning [2, 3] and the AFA concept. The integration works as follows. Basic reusable components of a family are derived from its architectural styles and are developed bottom-up as the family is initialized. Properties of these components are derived from the architectural styles and verified in the environments defined by these styles. The properties then serve as abstractions of the components in bottom-up verification of larger composite components. Top-down verification of a member system utilizes architectural styles to guide decomposition of properties of the system into properties of its components and formulation of environment assumptions of the component properties. The component properties are reused, if already verified; otherwise, they are verified top-down recursively. Architectural style driven property decomposition addresses formulation of component properties and their assumptions, and facilitates reuse of verified properties of reusable components. Additional reusable components may be introduced as the family evolves.

Preliminary case studies on web service based systems have shown that our approach is very effective in scaling verification: It achieved order-of-magnitude verification complexity reductions for non-trivial component-based systems and realized major verification reuse. The cost of our approach lies in configuring and evolving the AFA of a family and is amortized among member systems of the family. Further case studies are under way to evaluate if benefits obtained in verification and reuse justify the cost.

The rest of this paper is organized as follows. In Section 2, we introduce the concept of AFA and present an AFA for the domain of university information systems (UIS) based on web services. In Section 3, we discuss integrated bottom-up and top-down verification for an application family, which is illustrated with its application to the UIS domain in Section 5. In Section 6, we analyze the effectiveness and cost of integrated verification. We discuss related work in Section 7 and conclude in Section 8.

2 Application Family Architectures (AFAs)

AFAs extend the concept of software architectures [6, 7] and target model checking of component-based application families. An AFA for an application family is derived via domain analysis of this family. It captures common architectural styles of the systems in this family, which suggest properties that need to be verified on these systems and provide knowledge about possible composition environments for reusable components.

It also catalogs reusable components and their verified properties. An AFA is a 4-tuple, (computation model, component model, architectural style library, component library):

- The *computation model* defines the basic elements of a system: (1) the basic functional entities, (2) the interaction mechanism of these entities, and (3) the units of execution, and specifies the execution semantics in terms of these basic elements.
- The *component model* defines the concept of component, specifying the elements of a component: executable representation, interfaces (including functional interfaces and component properties), etc. It also defines the component composition rule.
- The *architectural style library* contains the common architectural styles that appear in systems of this family. An architectural style specifies the types of components that can be used in this style, the component interactions under this style, and a set of properties required by this style on these components and on their composition.
- The *component library* contains the reusable components that have been constructed for developing systems in this family. These components are reused in development of new systems. This library is expanded when new components are introduced.

2.1 AFA for University Information System

To illustrate this concept, we present an AFA for the domain of university information systems (UIS). A modern university is supported by many information systems such as the registration system, the library system, and the event ticketing system. Their central functionality is to process various electronic transactions. These systems are required to *correctly* process these transactions following the designated protocols.

Computation Model. An emerging trend is to develop information systems using web service technologies. Components of such systems are web services, implemented in program languages such as Java and C# or design-level executable languages such as Business Process Execution Language for Web Services (BPEL4WS) [8]. We formalize (with simplifications) the semantics of web service based systems as an Asynchronous Interleaving Message-passing (AIM) computation model. In this model, a system is a finite set of interacting processes. The processes interact via asynchronous message-passing. A system execution is an interleaving of the state transitions of these processes. In our previous work [9], we have developed the ObjectCheck toolkit which supports model checking of systems that follow the AIM semantics. We employ ObjectCheck as the model checker for verifying components and systems of the UIS family.

Component Model. Web services, the components in web service based systems, can be *primitive* (directly implemented) or *composite* (composed from simpler web services). Their interfaces are specified in XML-based interface specification languages, Web Service Definition Language (WSDL) [10] and Web Service Choreography Interface (WSCI) [11]. WSDL defines the message types supported by a web service. WSCI defines how the message types are interrelated in a choreographed interaction with the web service.

Component – A component C is a pair $(E,\ \{S\})$. E is the executable specification of C. Conceptually, E is a set of interacting AIM processes. (A primitive component

may contain multiple AIM processes.) Practically, E can be implemented in Java, C#, or BPEL4WS. $\{S\}$ is a set of services and each service S is a pair $(M,\ F)$ as follows.

- M is the messaging interface through which C provides the service S and requests the services necessary for providing S. M contains input and output message types and is specified in WSDL.
- F is the functional specification of the service S and is a pair *(provides, requires)*. The *provides* is a pair $(P(pro),\ A(pro))$ where $P(pro)$ is the temporal properties that define the service S and $A(pro)$ specifies the assumptions of $P(pro)$ on the components that request S. To provide S, C often requires other services. The *requires* is a set and each entry of the set is a pair $(P(req),\ A(req))$. $A(req)$ specifies the assumptions on a service S' required by C. $P(req)$ specifies the properties of C necessary for *enabling* the assumptions in $A(req)$, i.e., when C requests the service S', it must behave as $P(req)$ specifies. The properties and assumptions are formulated on the message types in M and are specified in WSCI.

This component definition facilitates assume-guarantee compositional reasoning by specifying properties with their assumptions and guides verification reuse by grouping properties and assumptions into the *provides* and *requires* format.

Component Composition – Composition of a set of components, $C_0, \ldots, C_{m-1}$, creates a composite component, $C = (E,\ \{S\})$, which provides services that aggregate the services provided by $C_0, \ldots, C_{m-1}$. Suppose the services $(M_0,\ F_0), \ldots, (M_{n-1},\ F_{n-1})$ of $C_0, \ldots, C_{m-1}$ are used to compose the service $(M,\ F)$ of C. (n can be bigger than m since multiple services of a component may be involved.)

- E is constructed from $E_0, \ldots, E_{m-1}$ by establishing mappings between incoming message types in M_i and outgoing message types in M_j, $0 \leq i, j < n$, in order to fully or partially satisfy the *requires* of F_i with the *provides* of F_j.
- M includes all message types in $M_0, \ldots, M_{n-1}$ that are needed for C to interact with its environment. F is defined on M. The *provides* of F is derived from the *provides* of one or several F_i's. The *requires* of F is derived from all entries in the *requires* of $F_0, \ldots, F_{n-1}$ that are not satisfied inside the composition.

F is verified on an abstraction of $C_0, \ldots, C_{m-1}$ constructed from $F_0, \ldots, F_{n-1}$. The abstraction includes all properties in the *provides* and *requires* of $F_0, \ldots, F_{n-1}$ whose assumptions are satisfied by the composition or the assumptions in the *provides* and *requires* of F. F is verified by checking the properties in the *provides* and *requires* of F on the abstraction. (See [12] for details of abstraction construction.)

Architectural Style Library. An architectural style is a triple, (component templates, service invocation graph, properties). The *component templates* are specified by component service interfaces which can be complete or partially defined, i.e., with partially defined messaging interfaces and *(provides, requires)* pairs. A component matches a component template if its interfaces match the interfaces of the component template. The *service invocation graph* is a directed graph that defines how the *requires* of the component templates are satisfied by the *provides* of other component templates. In a composite component following this style, the *provides* and *requires* of the subcomponents corresponding to the component templates must conform to the satisfaction

relationships. The *properties* are required to hold on a composite component following this style. They are formally defined on the interfaces of the component templates if the interfaces provide sufficient semantic information; otherwise, they are informally specified. A component is reusable if it matches a component template and its functionality is common across multiple composite components following this style.

The UIS architectural style library includes (but not limited to) the following styles:

- *Three-tier architecture.* (1) Component templates: The application logic, the business logic, and the database engine. The database engine is reusable. (2) Service invocation graph: This style features layered service invocation. The user logic invokes the business logic which, in turn, invokes the database engine. (3) Properties: The three components interact properly to ensure that their composition correctly processes each transaction received. The properties are informally specified due to insufficient semantic information about the transactions.
- *Agent-dispatcher.* (1) Component templates: A pool of agents and a dispatcher managing the agents. The dispatcher is a reusable component while the agents are different for different transactions, however, the agents conform to a partial interface whose *provides* is partially determined by the *requires* of the dispatcher. (2) Service invocation graph: The environment of a composite component following this style invokes the services of the dispatcher and agents. The dispatcher invokes the service of the agents. An agent provides services to the environment of the composite component and the dispatcher via the same messaging interface. (3) Properties: Upon a request from the environment if a free agent exists it must be dispatched. A dispatched agent is eventually freed. The properties are formally defined on the interfaces of the dispatcher template and the agent template.

Systems in the UIS family are transaction-oriented and circular service invocation is not permitted. The service invocation graphs are directed and acyclic. Dependencies between a service requester and its provider are captured in their *requires* and *provides*. Such dependencies do not cause circular reasoning due to the sequencing relationships among the messages of two interacting sub-components in executing a transaction.

Component Library. Basic reusable components of the UIS family, such as the database engine, are derived from its architectural styles. The desired properties of the database engine assert that it correctly handles each query. The properties have assumptions that databases are locked before and unlocked after they are queried and if multiple databases are accessed, they must be locked in a proper order to avoid deadlocks. The properties and their assumptions are parameterized by how many and what databases are accessed simultaneously. An instantiation of the properties for accessing a single database is shown in Figure 1. Space limitation prohibits showing the WSCI representations of the properties and assumptions. Instead, in Figure 1, the properties and assumptions are concisely specified in an ω-automaton based property specification language [9]. Each assertion is instantiated from a property template and is corresponding to an ω-automaton. Properties in this language are intuitive, for instance, the first assertion in Figure 1 asserts that after receiving a *lock* message, the database engine will eventually reply with a *locked* message. These specifications are translated from the WSCI specifications when the properties are verified using the ObjectCheck toolkit.

Provides:
P(pro):
After(Lock) **Eventually**(Locked); **Never**(Locked) **UnlessAfter**(Lock);
After(Locked) **Never**(Locked) **UnlessAfter**(Lock);
After(Unlock) **Eventually**(Unlocked); **Never**(Unlocked) **UnlessAfter**(Unlock);
After(Unlocked) **Never**(Unlocked) **UnlessAfter**(Unlock);
A(pro):
After(Lock) **Never**(Lock) **UnlessAfter**(Unlocked);
After(Locked) **Eventually**(Unlock); **Never**(Unlock) **UnlessAfter**(Locked);
After(Unlock) **Never**(Unlock)**UnlessAfter**(Locked);

Fig. 1. Properties of Database Engine

(For simplicity, only properties that are related to locking/unlocking are shown and the transaction identifiers are omitted from the messages.) The properties in *P(pro)* define the desired behaviors of the database engine. The assumptions in *A(pro)* specify the required behaviors of other components requesting the service. The database engine requires no other services. Besides the database engine, the agent-dispatcher style suggests the dispatcher service. These components are the initial components in the library.

2.2 Relationships of AFA to Verification

AFA extends the concept of software architectures to enable operational support for bottom-up component verification, top-down system verification, and their integration. The inclusion of a computation model and a component model in an AFA relates software architectures to component implementations and compositions, thus making the concept of software architectures operational for verification. The computation model guides the selection of model checkers. The component model provides compositional structures necessary for compositional reasoning. The architectural styles suggest component properties and how these properties are decomposed if needed.

3 Integrating Bottom-Up and Top-Down Verification

In this section, we present how the AFA concept facilitates bottom-up component verification, top-down system verification, and their integration. Our approach utilizes architecture styles captured by the AFA to guide property formulation and decomposition, and reduces complexities of verifying member systems based on compositional reasoning and on reuse of verified properties of reusable components available in the AFA.

3.1 Bottom-Up Component Verification in Family Initialization

As an application family is initialized, its basic reusable components are derived from its architectural styles. The properties of the components are formulated according to these styles. The assumptions of the component properties are also formulated according to how the components interact under the architectural styles. Derivation of reusable components and formulation of properties and assumptions requires manual efforts. Verification of the component properties follows the bottom-up approach developed in our previous work [12]. The properties of a primitive component, which is developed

from scratch, are directly model-checked. The properties of a composite component, instead of being checked on the component directly, are checked on its abstractions that are constructed from the verified properties of its sub-components. If the properties of the composite component cannot be verified on the abstractions, the abstractions are refined by introducing and verifying additional properties of the sub-components.

3.2 Top-Down System Verification in Member System Development

Development of a member system of an application family is top-down. The system is partitioned into its components which are reused from the component library, directly implemented, or partitioned recursively. A system is a composite component. Therefore, we discuss how a composite component is verified as it is developed top-down.

For a composite component following an architectural style, we integrate verification into its top-down development and utilize the architecture style to guide the decomposition of its properties into the properties of its sub-components.[1] We assume that the component interface has been designed. The properties of the composite component are formulated in the *(provides, requires)* format based on the interface and according to the architectural style. For architecture styles with informally specified properties, for instance, the 3-tier architecture, the property formulation requires manual efforts. The composite component is developed and verified using a top-down process as follows:

1. *Composite component layout.* The component is partitioned into its sub-components according to the architectural style. The sub-component interfaces are defined and the sub-component interactions are specified. This step requires manual efforts of the designers. The representation for sub-component interactions, for instance, High-level Message Sequence Charts (HMSC) [13] for the UIS family, are selected in conformance to the computation model and the component model of the family.
2. *Architectural style driven property decomposition.* The properties of the composite component are decomposed into the properties of its sub-components. The decomposition is guided by the architectural style and based on the sub-component interactions. How architectural styles guide property decomposition is discussed in detail in Section 4. The validity of the decomposition is established by ensuring that the properties of the sub-components imply the properties of the composite component and there exists no circular reasoning among sub-component properties. For a well-studied application domain, this step can be largely automated.
3. *Reuse or recursive development of sub-components.* The architectural style suggests whether a sub-component is reusable. There may be a set of components in the library which are reusable in a given sub-component role even though they are different in their interfaces or properties. A component is selected from the set based on their interfaces and properties. If no qualified component is found for a sub-component or it is suggested to be application-specific by the architectural style, it needs to be developed. If the sub-component is primitive, it is implemented, and its properties are verified through direct model checking of its implementation. If the

[1] A composite component may or may not follow an architecture style. A composite component following no style can be verified through compositional reasoning based on user-guided decomposition of properties of the composite component into properties of its sub-components.

sub-component is composite, it is developed and verified top-down. If it follows an architectural style, the top-down process discussed herein is applied recursively.

4. *Composition.* After all the sub-components are selected from the library or recursively developed, they are composed to construct the composite component by using the composition rule in Section 2.1 following the architectural style.

In each step of this process, failure to achieve the goal of the step will lead to revisions and re-executions of the previous steps or abortion of this process.

3.3 Bottom-Up Component Verification in Component Library Expansion

In the top-down development and verification of a member system, new components may be introduced. Some of these components are application-specific while the others are reusable. The properties of the reusable components have been established when the system is verified. These newly introduced reusable components may be further composed among themselves or with the existing reusable components to build larger reusable components bottom-up. Such a composite component is identified in the development of the member system and its sub-components together achieve a reusable functionality. The interface of the composite component is derived from the interfaces of its sub-components. The properties of the composite component are verified on its abstractions constructed from the properties of its sub-components. The sub-component properties are available from either verification of the member system or the component library. All these reusable components are then included into the component library.

3.4 Interactions of Bottom-Up and Top-Down Verification

Bottom-up and top-down verification are synergistic in their integration into the development lifecycle of an application family. Bottom-up component verification in family initialization provides the basis for verification reuse. Top-down member system development and verification expands the component library by introducing new reusable components and by enabling bottom-up construction and verification of larger reusable components. Component library expansion raises the level of component reuse and reduces the number of decompositions needed in top-down verification of a new system.

4 Architectural Style Driven Decomposition

The central step of top-down system verification is the architectural style driven property decomposition. In this step, the properties of a composite component (a system is a composite component) are decomposed into the properties of its sub-components based on the architectural style guiding the composition and on the sub-component interactions. For a well-studied domain, the decomposition procedure can be largely automated. How the decomposition procedure operates also depends on the representations of architectural styles, component interfaces, component interactions, and properties.

We present a decomposition procedure for the UIS family. (With slight modifications, this procedure can be generalized to many other transaction processing centric families.) Given a composite component C and a service $(M,\ F)$ that C is expected

to provide, the procedure decomposes the properties and assumptions in the *provides* and *requires* of F into the properties and assumptions of the sub-components of C following the architectural style of C. Properties and assumptions of a sub-component are grouped to define the services provided and required by the sub-component.

Under the UIS architectural styles, component interactions are transaction-oriented. To provide the service S, the sub-components C_0, ..., C_{n-1} of C interact following a transaction: a sequence of message communications through the messaging interfaces of C_0, ..., C_{n-1}. Component interfaces are service-oriented: a component provides a service and to provide the service, it requires services from other components.

We assume as C is designed, the interactions among C_0, ..., C_{n-1} are specified as a High-level Message Sequence Chart (HMSC) [13]. A HMSC allows branching upon different messages, repetitions of sub-sequences, and skips of sub-sequences. We also extend HMSCs by grouping the messages interactions among the sub-components according to service invocations. The message interactions for invoking a service are explicitly annotated. The external component that requires the service of C (denoted by *P-ENV*) and the set of the external components that provide the services required by C (denoted by *R-ENV*) are also represented in the HMSC. The message communications with *P-ENV* and *R-ENV* are derived from the *provides* and *requires* of F. Specifying HMSCs adds little extra costs to the design process of message-passing based systems.

The decomposition procedure for compositions whose service invocation graphs have tree structures is given as pseudo code in Figure 2. (Space limitation prohibits

```
procedure Decompose (style, comp-set, hmsc, current, parent)
begin
 if (current == P-ENV) then
   {children} = Find-Children (style, comp-set, hmsc, current);
   foreach child ∈ {children} do
     Decompose (style, comp-set, hmsc, child, current);
   endfor;
 elseif (current ∉ R-ENV) then
   provides = Derive-Provides-from-HMSC (hmsc, current, parent));
   {children} = Find-Children (style, comp-set, hmsc, current);
   foreach child ∈ {children} do
     req = Derive-Requires-from-HMSC (hmsc, current, child);
     requires = requires ∪ {req};
     Decompose (style, comp-set, hmsc, child, current);
   endfor;
   Attach-Service-to-Component (current, (provides, requires));
 endif;
end;
```

Fig. 2. The decomposition procedure

presenting the more complex decomposition procedure for compositions with directed acyclic service invocation graphs, which follows the same basic idea.) It inputs the architectural style guiding the composition, the set of sub-components represented by their messaging interfaces, the HMSC, the *current* sub-component whose service is to be derived, and the *parent* sub-component that requires the service of the *current* sub-component. The parent-children relationships among the sub-components are determined by the service invocation relationships among the sub-components defined in

the architectural style and the service annotations in HMSC. A component may appear in the children set of another components multiple times if it provides multiple services to its parent. The procedure is invoked with *P-ENV* as the *current* and NULL as the *parent* since *P-ENV* is the root of the transaction, and invokes itself recursively.

1. If *current* is *P-ENV*, the procedure locates all sub-components providing services to *P-ENV* and invokes itself recursively on each of these sub-components.
2. If *current* is not *P-ENV* and also not in *R-ENV*, the procedure first derives the *provides* of *current* from its interactions with its parent (the sub-component to which it provides the service). The procedure then finds all children of *current* (the sub-components that provide services to *current*), derives each entry of the *requires* of *current* from the interaction with each child, and invokes itself recursive on each child. The service, *(provides, requires)*, is then associated with *current*.
3. If *current* is in *R-ENV*, then nothing need be done.

Deriving the *provides* and *requires* of *current* from the HMSC is essentially projecting the HMSC onto *current* and the sub-components that interact with *current*. To derive the *provides*, the interactions of *current* with its parent are projected. To derive an entry of the *requires*, the interactions of *current* with one of its children are projected. The properties and assumptions in the *provides* and the *requires* are specified as WSCI processes. A WSCI process is a simple state machine that captures the behaviors of a sub-component as specified in the HMSC: receiving incoming messages and responding with outgoing messages. The derivation algorithm is straightforward. Receiving and sending messages in the HMSC is captured as atomic messaging activities in the WSCI process. Sequencing relationships among messages in the HMSC are captured by sequence activities in the WSCI process. Branchings according to different messages received in the HMSC are captured by choice activities in the WSCI process.

Space limitation precludes presentation of a detailed correctness proof of the decomposition procedure. The intuition is as follows. The procedure always terminates since it goes through each component following an order determined by the architectural style. The procedure ensures that the composition of the derived services of the sub-components implies the service of the composite component. The *requires* of *P-ENV* is satisfied by the *provides* of its children sub-components whose *requires* are satisfied by their children recursively. The *requires* of the sub-components that interact with *R-ENV* are satisfied by the *provides* of *R-ENV*. Therefore, the composite provides the *requires* of *P-ENV* if *R-ENV* provides the *requires* of the composite. In addition, the acyclic service invocations among the sub-components and the sequencing relationships among the messages of two interacting sub-components prevent circular reasoning.

5 Integrated Bottom-Up and Top-Down Verification of UIS

5.1 Bottom-Up Component Verification in Family Initialization

As the UIS family is initialized, its architectural styles suggest two reusable components: the database engine and the dispatcher. Verification of database engines is out of the scope of this paper. We assume that the properties of the database engine hold. The

Provides:
P(pro): **After**(Login) **Eventually**(TryLater + Dispatch); **Never**(TryLater + Dispatch) **UnlessAfter**(Login);
After(TryLater + Dispatch) **Never**(TryLater + Dispatch) **UnlessAfter**(Login);
A(pro): (Empty)
Requires:
A(req): **After**(Dispatch) **Eventually**(Free); **Never**(Free) **UnlessAfter**(Dispatch);
After(Free) **Never**(Free) **UnlessAfter**(Dispatch);
P(req): **After**(Dispatch) **Never**(Dispatch) **UnlessAfter**(Free);

Fig. 3. Properties of Dispatcher

dispatcher is a primitive component. Its properties and their assumptions are shown in Figure 3. The properties in *P(pro)* and *P(req)* are checked on the dispatcher under a non-deterministic environment whose interface complements the interface of the dispatcher and which is constrained by the assumptions in *A(pro)* and *A(req)*. The properties were verified in 0.9 seconds and 0.16 megabytes which are order-of-magnitude lower than the time and memory usages for verifying a system utilizing the dispatcher service (see Section 6). No composite reusable components are introduced in family initialization.

5.2 Top-Down System Verification in Member System Development

We illustrate top-down system verification through verifying the registration system from the UIS family. The registration system is structured following the 3-tier architecture. It consists of three components: the application logic, the business logic, and the database engine. The interactions among these components and the environment of the registration system are captured by a HMSC. Upon a login request, the system execution may take three branches: (1) log the user in; (2) reject the user; (3) ask the user to try later. For illustration purposes, the first two branches are shown in Figure 4 as two

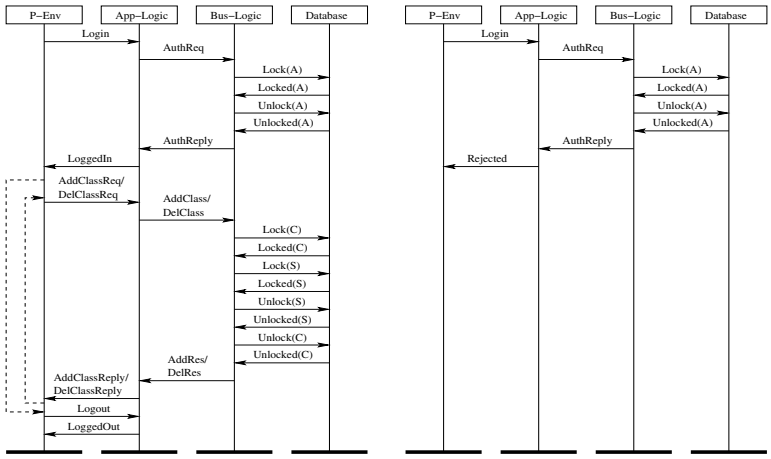

Fig. 4. A flattened view of the HMSC for component interactions under the 3-tier architecture

MSCs with the following extensions: The forward dashed arrow denotes the skip of a sub-sequence and the backward dashed arrow denotes the repetition of a sub-sequence. For instance, after a user logs in, she may or may not add or delete classes, and she may add or delete multiple classes. In Figure 4, service annotations that group messages into service invocations are not shown for simplicity. The message interactions between the application logic and the business logic are grouped into two service invocations: one for authentication and the other for adding or deleting classes. Similarly, the message interactions between the business logic and the database engine are grouped into two service invocations: one for access to the authentication database and the other for simultaneous access to the class database and the student database.

The 3-tier architecture requires verifying that the registration system follows the designated message sequences for a registration transaction when interacting with a well-behaved user. Essentially, we verify that the system interacts with such a user following the message sequences between *P-ENV* and the application logic in Figure 4.

The properties of the registration system can be automatically derived from the HMSC as follows. A WSCI process is created from the HMSC and captures the messages from *P-ENV* to the system, the response messages of the system, and the sequencing relationships among the messages observed by the system. Essentially, the WSCI process is obtained from the HMSC by projecting the interactions between the system and *P-ENV* onto the system. Space limitation prohibits showing the WSCI process. Instead, its formal translation is shown in Figure 5. The temporal predicates in *P(pro)* encode the WSCI process, i.e., capturing the temporal relationships among the messages, for instance, the first three predicates in *P(pro)* capture the temporal relationships between *Login* and *LoggedIn*, *Rejected*, and *TryLater*. *A(pro)* is derived from the HMSC by projecting the interactions of *P-ENV* and the system onto *P-ENV*. For instance, the first predicate in *A(pro)* specifies an assumption on *P-ENV* that it never sends an *AddClassReq*, *DelClassReq*, or *Logout* message unless after it receives a *LoggedIn* message. The properties in *P(pro)* and the assumptions in *A(pro)* are interdependent and together they capture the message interactions between *P-ENV* and the system. Since the registration system requires no other services, its *requires* is empty.

Provides:
P(pro):
After(Login) **Eventually**(LoggedIn+Rejected+TryLater);
Never(LoggedIn+Rejected+TryLater) **UnlessAfter**(Login);
After (LoggedIn+Rejected+TryLater) **Never** (LoggedIn+Rejected+TryLater) **UnlessAfter**(Login);
After(AddClassReq) **Eventually**(AddClassReply); **Never**(AddClassReply) **UnlessAfter**(AddClassReq);
After(AddClassReply) **Never**(AddClassReply) **UnlessAfter**(AddClassReq);
After(DelClassReq) **Eventually**(DelClassReply); **Never**(DelClassReply) **UnlessAfter**(DelClassReq);
After(DelClassReply) **Never**(DelClassReply) **UnlessAfter**(DelClassReq);
After(Logout) **Eventually**(LoggedOut); **Never**(LoggedOut) **UnlessAfter**(Logout);
After(LoggedOut) **Never**(LoggedOut) **UnlessAfter**(Logout);
A(pro):
Never(AddClassReq+DelClassReq+Logout) **UnlessAfter**(LoggedIn);
After(AddClassReq) **Never** (AddClassReq+DelClassReq+Logout) **UnlessAfter**(AddClassReply);
After(DelClassReq) **Never** (AddClassReq+DelClassReq+Logout) **UnlessAfter**(DelClassReply);
After(LoggedIn) **Eventually** (Logout); **After**(Logout) **Never**(AddClassReq+DelClassReq+Logout);

Fig. 5. Properties of Registration System

The properties of the registration system are decomposed into the properties of its sub-components by the decomposition procedure in Section 4. The procedure starts with *P-ENV* and invokes itself recursively on the three sub-components of the system following the service invocation graph of the 3-tier architecture. The first component whose properties are derived is the application logic. The derived properties and assumptions of the application logic are shown in Figure 6. The application logic

Provides: (same as the *provides* in Figure 5.)
Requires 1:
A(req):
After(AuthReq) **Eventually**(AuthReply); **Never**(AuthReply) **UnlessAfter**(AuthReq);
After(AuthReply) **Never**(AuthReply) **UnlessAfter**(AuthReq);
P(req): After(AuthReq) **Never**(AuthReq) **UnlessAfter**(AuthReply);
Requires 2:
A(req):
After(AddClass) **Eventually**(AddRes); **Never**(AddRes) **UnlessAfter**(AddClass);
After(AddRes) **Never**(AddRes) **UnlessAfter**(AddClass);
After(DelClass) **Eventually**(DelRes); **Never**(DelRes) **UnlessAfter**(DelClass);
After(DelRes) **Never**(DelRes) **UnlessAfter**(DelClass);
P(req):
After(AddClass) **Never**(AddClass+DelClass) **UnlessAfter**(AddRes);
After(DelClass) **Never**(AddClass+DelClass) **UnlessAfter**(DelRes);

Fig. 6. Properties of Application Logic

provides the registration service to *P-ENV*. The procedure derives the *provides* interface of the application logic from its message interactions with *P-ENV*. The *provides* interface is derived by projecting the message interactions between *P-ENV* and the application logic and it is essentially the same as the *provides* interface of the registration system. The procedure determines from the HMSC that to provide the registration service, the application logic requires two services from the business logic: one for authentication and the other for adding or deleting classes. The corresponding *requires* entry for each of the two services is derived from the message interactions with the business logic. The *A(req)* is derived by projecting the message interactions onto the business logic while *P(req)* is derived by projecting the message interactions onto the application logic.

Following the service invocation relation between the application logic and the business logic, the decomposition procedure is invoked to derive the properties of the business logic. Based on the HMSC service annotations, the procedure is invoked for each service that the business logic provides. The properties are shown in Figure 7, capturing the services provided to the application logic and required from the database engine.

The database engine processes two types of service invocations: access to the authentication database and simultaneous access to the student and class databases. The properties and assumptions in the *provides* of the database engine are the same as the assumptions and properties in the *requires* of the business logic. The database engine has no *requires*. The properties and assumptions of the two service invocations differ since they are instantiated differently. The database engine introduced in the family initialization is selected for reuse since it has a matching messaging interface and its properties (or assumptions), instantiated by how many and what databases are accessed, imply (or are implied by) the properties (or assumptions) derived in the top-down decomposition.

/* Service 1 */
Provides:
P(pro) (or **A(pro)**, respectively) is the same as *A(req)* (or *P(req)*) of *Requires 1* of Application Logic.
Requires:
A(req) (or **P(req)**, respectively) is same as *P(pro)* (or *A(pro)*) of *Provides* of the DB engine in Figure 1.
/* Service 2 */
Provides:
P(pro) (or **A(pro)**) is same as *A(req)* (or *P(req)*) of *Requires 2* of Application Logic.)
Require:
A(req):
After(Lock(C)) **Eventually**(Locked(C)); **Never**(Locked(C)) **UnlessAfter**(Lock(C));
After(Locked(C)) **Never**(Locked(C)) **UnlessAfter**(Lock(C));
After(Lock(S)) **Eventually**(Locked(S)); **Never**(Locked(S)) **UnlessAfter**(Lock(S));
After(Locked(S)) **Never**(Locked(S)) **UnlessAfter**(Lock(S));
After(Unlock(S)) **Eventually**(Unlocked(S)); **Never**(Unlocked(S)) **UnlessAfter**(Unlock(S));
After(Unlocked(S)) **Never**(Unlocked(S)) **UnlessAfter**(Unlock(S));
After(Unlock(C)) **Eventually**(Unlocked(C)); **Never**(Unlocked(C)) **UnlessAfter**(Unlock(C));
After(Unlocked(C) **Never**(Unlocked(C)) **UnlessAfter**(Unlock(C));
P(req):
After(Lock(C)) **Never**(Lock(C)) **UnlessAfter**(Unlocked(C));
After(Locked(C)) **Eventually**(Lock(S)); **Never**(Lock(S)) **UnlessAfter**(Locked(C));
After(Lock(S)) **Never**(Lock(S)) **UnlessAfter**(Locked(C));
After(Locked(S)) **Eventually**(Unlock(S)); **Never**(Unlock(S)) Unless**After**(Locked(S));
After(Unlock(S)) **Never**(Unlock(S)) **UnlessAfter**(Locked(S));
After(Unlocked(S)) **Eventually**(Unlock(C)); **Never**(Unlock(C)) **UnlessAfter**(Unlocked(S));
After(Unlock(C)) **Never**(Unlock(C)) **UnlessAfter**(Unlocked(S))

Fig. 7. Properties of Business Logic

The structure of the application logic follows the agent-dispatcher style. For each user request, the dispatcher dispatches an agent to serve the user if there exists a free agent; otherwise, it asks the user to try later. The properties of the application logic are decomposed into the properties of the dispatcher and the agents. Based on the derived properties for the dispatcher, the dispatcher that has been introduced and verified when the UIS family is initialized is selected for reuse. The *provides* and *requires* of the agents are largely the same as those of the application logic except the properties and assumptions that are related to agent dispatching, which are shown in Figure 8.

Provides:
P(pro):
After(Dispatch) **Eventually**(LoggedIn+Rejected); **Never**(LoggedIn+Rejected) **UnlessAfter**(Dispatch);
After(LoggedIn+Rejected) **Never**(LoggedIn+Rejected) **UnlessAfter**(Dispatch);
After(Dispatch) **Eventually**(Free); **Never**(Free) **UnlessAfter**(Dispatch);
After(Free) **Never**(Free) **UnlessAfter**(Dispatch);
A(pro): After(Dispatch) **Never**(Dispatch) **UnlessAfter**(Free);

Fig. 8. Properties and Assumptions of Agents Related to Dispatching

The business logic is partitioned into the authentication processor and the registration processor. Each implements a service of the business logic shown in Figure 7.

5.3 Bottom-Up Component Verification in Component Library Expansion

As the registration system is developed, the authentication processor is introduced in the business logic layer and it interacts with the database engine to provide the user

authentication. The two components is composed bottom-up to build an authentication service that processes authentication requests and replies to these requests. The desired properties of the authentication service is shown in Figure 9. The properties, instead

Provides:
A(req):
After(AuthReq) **Eventually**(AuthReply); **Never**(AuthReply) **UnlessAfter**(AuthReq);
After(AuthReply) **Never**(AuthReply) **UnlessAfter**(AuthReq);
P(req): After(AuthReq) **Never**(AuthReq) **UnlessAfter**(AuthReply);

Fig. 9. Properties of Authentication Service

of being checked directly on the authentication service, is checked on its abstraction. The abstraction is constructed from the verified properties of the authentication processor and the database engine. The properties of the authentication processor have been established in the top-down system verification while the properties of the database engine have been established in the family initialization. The introduction of the authentication service suggests the introduction of a new architectural style: 3-tier architecture with authentication, as shown in Figure 10. In development of new systems such as the

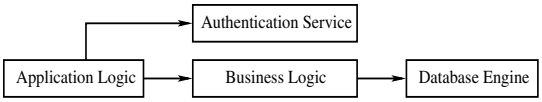

Fig. 10. 3-tier architecture with authentication

library system and the ticket sale system, the new style can be selected to structure these systems and, therefore, facilitate reuse of the authentication service and its properties.

6 Effectiveness and Cost of Integrated Verification

Our integrated approach has major potential for improving reliability of a component-based application family. It enables effective verification of member systems of the family by greatly reducing verification complexities of the systems and facilitating verification reuse. Direct verification of the properties of the registration system with a configuration of 3 concurrent users and 2 agents takes 7034.27 seconds and 502.31 megabytes and it does not scale to large configurations. In verifying the same system with our approach, only the properties of the agent, the authentication processor, and the registration processor must be verified and the properties of other components are reused. The time and memory usages for verifying these components are shown in Table 1. It can be observed that our approach achieves order-of-magnitude reduction in verification time and memory usages. Our approach scales to member systems of large configuration via systematic partition of a system into components of manageable size.

Table 1. Verification time and message usage

	Agent	Authentication Processor	Registration Processor
Time (Seconds)	0.75	0.1	4.09
Memory (MBytes)	0.29	0.31	0.31

The cost of our approach lies in initializing, maintaining, and evolving the AFA: identifying and capturing architectural styles, bootstrapping the component library, and expanding the library. The cost, however, is amortized across the member systems of an application family. Architectural style driven property decomposition procedures are often reused across multiple application families, for instance, the decomposition procedure in Section 4 can be reused across many transaction processing centric families. We are currently conducting further case studies on families of web service based systems and embedded systems to evaluate whether the cost of applying our approach can be justified by the benefits obtained in system verification and verification use.

7 Related Work

The concept of AFAs extends the concept of software architectures [6, 7] and targets verification of families of component-based systems. Space limitation prohibits full coverage of related work on software product families. The Product Line Initiative [14] at SEI focuses on design and implementation issues for software product families. Our work differentiates by focusing systematic verification of software application families.

Pattern reuse is often conducted at two levels: design level and architectural level. Design patterns [15] are concerned with reuse of programming structures at the algorithmic or data structure level. Architectural styles (a.k.a., architectural patterns) [6, 7] are concerned with reusable structural patterns of software systems with respect to their components. Architectural styles have been applied in system design, documentation, validation, etc. Our research utilizes architectural styles of a component-based application family to facilitate component property formulation and decomposition.

A major challenge to assume-guarantee compositional reasoning is formulation of component properties and their environment assumptions. There are approaches [4, 5] to automatic generation of assumptions for safety properties of components. Our approach addresses this challenge via architectural style guided property formulation in bottom-up component verification and via architectural style driven property decomposition in top-down system verification. It handles both safety and liveness properties and complements automatic assumption generation for safety properties of components.

8 Conclusions and Future Work

We have presented a novel approach to formal verification of software application families. This approach synergistically integrates bottom-up component verification and top-down system verification into the development lifecycle of software application

families. Its application to the UIS family has shown that it enables verification of non-trivial systems and reuse of major verification efforts. Currently, we are conducting further case studies to evaluate whether the benefits obtained by our approach in system verification and verification reuse can justify the cost of our approach.

References

1. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (1999)
2. Chandy, K.M., Misra, J.: Proofs of networks of processes. IEEE TSE **7**(4) (1981)
3. Jones, C.B.: Development methods for computer programs including a notion of interference. PhD thesis, Oxford University (1981)
4. Gannakopoulou, D., Pasareanu, C., Barringer, H.: Assumption generation for software component verification. In: ASE. (2002)
5. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional reasoning by learning assumptions. In: CAV. (2005)
6. Perry, D., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT SEN **17**(2) (1992)
7. Shaw, M., Garlan, D.: Software Architecture: Perspective on An Emerging Discipline. Prentice Hall (1996)
8. IBM: Business Process Execution Language for Web Services (BPEL4WS), Ver. 1.1. (2003)
9. Xie, F., Levin, V., Kurshan, R.P., Browne, J.C.: Translating software designs for model checking. In: FASE. (2004)
10. W3C: Web Services Description Language (WSDL), Ver. 1.1. (2001)
11. W3C: Web Service Choreography Interface (WSCI), Ver. 1.0. (2002)
12. Xie, F., Browne, J.C.: Verified systems by composition from verified components. In: ESEC/SIGSOFT FSE. (2003)
13. ITU: Rec. Z.120, Message Sequence Chart. (1999)
14. Clements, P.C., Northrop, L.M.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Object-Oriented Software. Addison-Wesley (1994)