Extending xADL with Statechart Behavioral Specification

Leila Naslavsky, Lihua Xu, Marcio Dias, Hadar Ziv, and Debra J. Richardson
Department of Informatics – School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 - USA
{lnaslavs, lihuax, mdias, ziv, djr}@ics.uci.edu

Abstract

Architecture-based analysis and testing of systems plays a key role in increasing their dependability. In order to perform those activities, both structural and behavioral architecture descriptions are needed. In most ADLs, support for representing dynamic behavior at the architectural level is either unavailable or is available only using the particulars of that ADL, thereby limiting its usefulness. xADL, an XML-based architecture representation, provides only structural architectural descriptions, not enough to perform analysis of dynamic behavior. This paper builds upon xADL and an existing analysis tool (Argus-I) to create an approach to xADL's current structural description features, augmenting it with behavioral specifications, namely statecharts. We believe that architectural analysis tools should be independent of their ADLs. Adding explicit support for behavioral description in xADL is the first step toward proper integration of flexible representations of architectures with flexible capabilities for their analysis and test.

1. Introduction

The main objective of this work is to bring together two lines of research in software architecture that are currently lacking in their integration, thereby enhancing both areas as well as overall system dependability. The first research area is architectural analysis and testing: To predictably and reliably build complex systems by composing components, components must be analyzed not only independently but also in the context of other components, connectors, and configurations. Analysis and testing of system architectures are fundamental to system dependability, since components, connectors, and their configurations are better understood and intellectually tractable. Architectural analysis should include both static structural and dynamic analysis techniques; architectural testing should include testing of components and their interrelationships as well as regression testing, residual testing, and so on.

The second research area is architecture definition languages (ADLs), specifically recent efforts toward flexible and extensible architectural representations. To support sophisticated architecture-based analysis, both the static structure as well as dynamic behavior of systems must be specified. Unfortunately, many ADLs such as ACME [1], C2SADL [2], Aesop [3] and UniCon [4] have limited, indirect or no support for behavior description. Those that support behavioral description are augmented with analysis tools such as constraint checkers and simulators. Each analysis tool has its advantages and disadvantages and it would be useful to explore their abilities independently of the ADL they operate over. xADL [5][6][7] is an effort to move away from proprietary ADL solutions by utilizing a standard and extensible XML-based representation for software architectures. Regrettably, xADL still lacks support for behavioral descriptions as well as any accompanying analysis tools.

It is worth noting that Statecharts in UML are normally associated to classes (design elements) in software systems, instead of software architecture elements. Thus, when considering architecture-based software dependability via analysis and testing, we need to associate Statecharts with architecture elements. xADL certainly provides the platform where we can adapt/extend Statecharts to capture the behavioral description in software architecture. By extending xADL with Statecharts, we can explore and perform both structural and behavioral analysis over software architecture, in order to enhance system dependability.

In earlier work, our research group has developed a platform for architectural analysis named Argus-I ("All Seeing" Architectural Analysis [8][9]). It is a set of specification-based analysis tools focusing on both the component and architecture levels of C2 style architectures. In Argus-I, structural and behavioral

analyses are accomplished by a combination of static and dynamic techniques. The key underlying representation used to support dynamic analysis is an extension of UML Statecharts [10] for the C2 architectural style. One weakness of the Argus-I environment is its reliance on a specific ADL and its representations; in the case of Statecharts, Argus-I relies on the extension of Statecharts with specific regards to the C2 style. This limits Argus-I not only to a specific representation of Statecharts but also to the C2 style and its specific ADL representation.

We contend that Argus-I could be re-factored to support xADL, thus supporting architecture-based analysis of a variety of ADLs. Unfortunately, the lack of semantic definition of component behavior in xADL limits the prospects of architecture-based analysis. We therefore extend xADL with statechart representations of the dynamic behavior of architecture components.

1.1 Contributions and Status

This paper offers the following contributions:

- Define a conceptual model and XML schema for Statecharts to be used for dynamic modeling at the architectural level. This work is done and its outcome described in Sections 3.1 and 3.2 below.
- Augment component-behavior modeling in xADL to support the XML representation of Statecharts as discussed above. This work is done and its outcome described in Section 3.3 below.
- Re-factor Argus-I to utilize this representation of Statecharts instead of the current XMI one. This is straightforward implementation work, expected to be accomplished soon.
- Extend both Argus-I and xADL to further enhance their integration. The work described here is a first step toward integration of architecture analysis and test capabilities with flexible ADLs; additional future steps are discussed in Section 4 below. We are especially interested in improving specific test techniques – such as regression test and residual test of system architectures – using the new XML representations of Statecharts.

2. Background and Related Work

Research in software architectures and languages for their definition is directed at formal modeling notations as well as analysis and development tools operating on the high-level structure (i.e., architecture) of systems, while leaving out any implementation details of source-code or deployment-level modules.

xADL 2.0 is an XML-based representation for software architectures, designed for developing various architecture types by adopting XML schemas and their extensibility as the basis for development. xADL 2.0 is an application of xArch [7], an extensible XML-based representation for software architectures. xADL extends xArch with support for architecture-level configuration management concepts, architectural prescription, a types-and-instances model, and more. However, the lack of semantic definition in xADL for component behavior limits architecture-based analysis and is one of the motivations for our work.

Argus-I is a comprehensive set of specification-based analysis tools focusing on both the component and architecture levels. Both structural and behavioral analyses are supported synergistically. Static analysis can, for instance, detect incompatibility between the data exchanged between components and verify architectural adherence to design heuristics and style rules. Dynamic analysis may be used to reveal defects in dynamic component interaction and communication behavior between components [8][9].

Regression Testing is performed on a modified version of a software system after changes have been made to the system to make sure no new errors were introduced. RTMC (Regression Testing via Model Checking) [11] is an ongoing project in our research group. The main thrust of RTMC is generation of regression test cases based on formal specifications derived early in the development lifecycle instead of directly from software source code. We use model checker as a part of our test generation tool, taking as input the differences between the original model and the modified version as inputs and producing test sequences for regression testing. RTMC is, therefore, an architecture-based testing tool, requiring not only a flexible representation to model software architectures independently from their ADLs, but also to describe the changes made to the system, both structural and behavioral changes. Augmenting xADL with statechart descriptions of dynamic behaviors would help fulfill the latter requirement. Specifically, changes to the XML representation of a given statechart can be used to guide the generation of regression test cases.

Residual testing, as introduced by Pavlopulou and Young [12], allows software developers to cope with software deployed without full testing coverage. It tests deployed software with respect to remaining test obligations (the residue). It employs selective

software instrumentation and monitoring techniques to collect usage data for the purpose of software quality assurance and evolution. In earlier work [13], we suggested that residual testing be used to monitor additional test coverage – and therefore satisfaction of additional test obligations – by consolidating executions of multiple deployments of the application. The resulting information is also useful for fixing or improving the product.

Our approach to residual testing explores test coverage criteria defined over call-graphs. We choose this coarse-grain level of abstraction rather than a fine-grain approach (such as statement execution) since (1) deployed applications should be tested at a higher level of abstraction to avoid undue performance degradation, and (2) this level of abstraction is closer to software design, which supports our goal of verifying specified behavior. The ever-increasing complexity of systems further amplifies the need for this kind of testing activity at an even higher level of abstraction, namely component and architectural Components are reused in unpredictable environment so fully testing them is also a very difficult (if not impossible) activity. Ensuring a component's structural coverage does not eliminate the need for verifying behavioral consistency. Thus, the representation of the component's dynamic behavior is key to supporting residual testing with behavioral verification at the architecture level.

3. Components Behavior in xADL

We represent component behavior in xADL by expressing the behavior description in an XML schema format and adding this information to one of xADL's schemas already defining a component. xADL schemas are in turn defined as extensions to core xArch schema. More precisely, our component behavior description is based on an extension of the Statechart model from the "UML Interchange Language" and "UML Semantics"[14], which provide a detailed class diagram for UML statecharts, as well as their semantics.

3.1. Statechart Conceptual Model

In order to show how statecharts can be used to describe software-component behavior, we define a conceptual model – a variation of state machines in the UML interchange metamodel – as described below. The reader should refer to Figure 1 for a UML object model of the elements of a statechart and their relationships.

Action: The specification of an executable statement that forms an abstraction of a computational procedure. This procedure would cause a change in the state of the component. An action can be realized as a message being sent to other components, or by a change in the internals characteristics of the component.

ActionSequence: A collection of actions. In the model, the ActionSequence is a kind of Action as well as a composition of ordered actions.

CompositeState: A state that contains other state vertices (see the StateVertex), which can be either a state or a pseudo state.

Event: The *type* of something that happens in the system (an observable occurrence) that has no duration, i.e., is instantaneous. From the model, it can be observed that an event (type) is aggregated to one or more transitions.

ElseEvent: A specialization of an Event; it is a type of event that might enable its corresponding transition in case no other outgoing transition was enabled from the current state when an event occurred.

ElseGuard: A specialization of a Guard. This kind of guard evaluates to true in case all guards in all outgoing transitions from the current state vertex evaluate to false. At most one guard of this type can exist in an outgoing transition per state vertex.

ExceptionGuard: Another specialization of a Guard. This kind of guard is used to define an internal error condition not identifiable with a boolean expression.

FinalState: A specialization of a state, FinalState indicates that the activities of the enclosing state have finished.

Guard: Offers fine-grain control over whether or not a transition should be fired. It has a boolean expression as attribute and is attached to a transition.

PseudoState: An extension of a StateVertex; it comprises the special pre-defined types of transient vertices. It has an attribute ("kind") that indicates its type among the possible pre-defined types.

State: An abstract type, which will be realized in a statechart as a CompositeState, a FinalState or a SimpleState. It has tree associations to actions named "entry" (executed upon entering the state), "exit" (executed upon leaving the state) and "doActivity" (executed while in the state).

StateChart: The main element in the model. It has a composition relationship to at most one state element, named "top" and a composition relationship to unlimited transition elements.

StateVertex: An abstraction of node in a statechart.

It can have many incoming and outgoing transitions.

SimpleState: A state specialization that does not have sub-states (see CompositeState).

Transition: A transition represents the possibility of a state change, in the model it is composed by at most one guard, unlimited number of events and at most one action. The relationship to the action is named “effect”.

TrueEvent: A specialization of an Event that enables all its corresponding transitions as soon as the “source” statevertex in the association with that transition is reached.

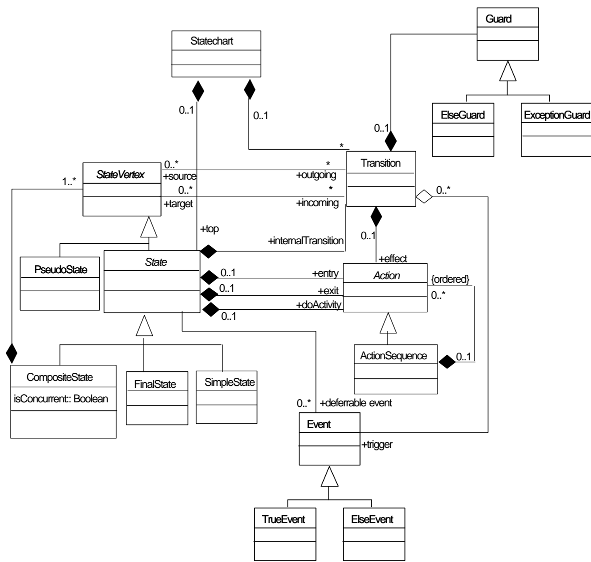

Figure 1. StateChart Conceptual Model

```
<xs:element name = “StateChart”>
  <xs:complexType>
      <xs:sequence>
    <xs:element name=”transitions” type=TransitionType”
          maxOccurs=”unbounded”/>
          <xs:element name=”top” type=”StateType”/>
      </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 2, XML-schema of element StateChart

3.2. Statechart XML-Schema

Given the conceptual model, we need to express it as an XML-Schema. First, we show how a composition relation is expressed in an XML schema. The main element, for example, is stateChart, including the state and transition elements as discussed in Section 3.1 (Figure 2). The statechart is considered the root element in the XML schema. The root element is a complexType because it is composed by other elements. These elements are the “transitions” and the “top” state. The transitions element is of type “TransitionType”, also defined in the schema, its maximum occurrence is unbounded to express this same property from the model (“*”). The “top” element is of type “StateType”, also defined in the schema.

```
<xs:complexType name=”StateVertexType”>
   <xs:sequence>
 <xs:element name=”outgoing” type=”TransitionType” />
 <xs:element name=”incoming” type=”TransitionType” />
   </xs:sequence>
</xs:complexType>
<xs:complexType name =”StateType”>
 <xs:complexContent>
 <xs:extension base=”StateType”>
 <xs:sequence>
        <xs:element name=”internal” type=”TransitionType” />
        <xs:element name=”entry” type=”ActionType” />
        <xs:element name=”exit” type=”ActionType” />
        <xs:element name=”doActivity” type=”ActionType” />
        <xs:element name=”deferrable” type=”EventType” />
 </xs:sequence>
 </xs:complexContent>
</xs:complexType>
```

Figure 3, Inheritance and Association

We show the most important part of the model that comprises inheritance and association and explain how those are expressed in the XML schema. Figure 3 shows the partial XML schema for abstract statevertex, the state, and the compositestate (as discussed in Section 3.1). The statevertex is defined as a “complexType” and its associations to the transition are elements of type “TransitionType,” named after the role played by the transition. The inheritance between statevertex and state is described by defining the StateType as an extension element with the StateVertexType as its base. Additional elements and their types are also illustrated in the schema.

3.3. Augmenting component behavior in xADL

After defining statechart semantics as XML schema, we need to describe how this information can be integrated into a xADL component type. A new type of component needs to be defined in xADL; this can be done in one of three ways. The first is defining a new complex type to describe the connection between the xADL’s component type and the statechart. The second is to completely replace the component type described in xADL with a new one. This new component type would have one more element to address the statechart description. The third option is to define a new component type that extends xADL’s component type by adding an element with the behavior description.

Since xADL is not bound to a particular behavior description formalism (there are other possibilities,

such as Labeled Transition System, Posets, Petri nets, and so on), it is not appropriate to define, a priori, specifically which kind of formalism will be used to describe the component behavior. Additionally, as discussed above, xADL adopts XML schemas and their extensibility mechanisms to define modularly extensible features. Thus, we propose to extend the original component type, which can be implemented as shown in Figure 4:

```
<xsd: complexType name="ourComponentType">
  <xsd:complexContent>
    <xsd:extension base="ComponentType">
      <xsd: sequence>
        <xsd:element name="Behavior" type="StatechartType"/>
      </xsd:sequence>
      </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
```

Figure 4, Augmenting xADL

We extend the component type defined in xADL with the new component type *ourComponentType*, and add the attribute *Behavior* of type *StatechartType*.

4. Future work and Conclusions

Architectural descriptions should be "open" with respect to analysis and test tools. It should be possible to integrate and apply a given analysis or test capability to a given architectural representation with utmost flexibility. Thus, the architectural description language should be able to interact with any analysis technique that works with information in the architectural specifications. We believe that our work can be adapted/extended to be useful in the field of architecture-based analysis and testing. For instance: reusing Argus-I to do analysis of xADL; integrating xADL XML-schema with the efforts in architecture-based regression testing and residual testing.

We are considering extending our work to support other formalisms for architecture specification, such as Labeled Transition System, Petri nets, and so forth; and pursuing the possibility of using these extended architecture description for analysis and testing of software systems. We believe that an integrated set of support capabilities for architecture and component specification and analysis will greatly enhance the quality of systems developed in the architecture-based and component-based software engineering paradigms.

Acknowledgments

The authors would like to acknowledge and thank the ROSATEA research group in the School of Information and Computer Science at the University of California, Irvine for their valuable insights.

References

1. Garlan, D., R.T. Monroe, and D. Wile *Acme: Architectural Description of Component-Based Systems*, in *Foundations of Component Based Systems*, C.U. Press, Editor. 2000.
2. Medvidovic, N., D. Rosenblum, and R. Taylor. *A language and environment for architecture-based software development and evolution.* in *21st ACM International Conference on Software Engineering*. 1999.
3. Garlan, D., R. Allen, and J. Ockerbloom. *Exploiting Style in Architectural Design Environments*. in *Symposium on the Foundations of Software Engineerng*. 1994.
4. Unicon. *http://www-2.cs.cmu.edu/afs/cs/project/vit/www/unicon/index.html*.
5. Dashofy, E.M., A.v.d. Hoek, and R.N. Taylor. *An Infrastructure for the Rapid Development of XML-based Architecture Description Languages*. in *24th International Conference on Software Engineering (ICSE 2002)*. 2002. Orlando, Florida: ACM.
6. Medvidovic, N., *A Classification and Comparison Framework for Software Architecture Description Languages*. 1997, University of California, Irvine.
7. Dashofy, E.M., A.v.d. Hoek, and R.N. Taylor. *A Highly-Extensible, XML-Based Architecture Description Language*. in *Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*. 2001. Amsterdam, The Netherlands.
8. Vieira, M., M. Dias, and D.J. Richardson. *Describing Dependencies in Component Access Points*. in *4th Workshop on Component Based Software Engineering, held in conjunction with the 23rd International Conference on Software Engineering (ICSE '01)*. 2001. Toronto, Canada.
9. Dias, M., M. Vieira, and D.J. Richardson. *Analyzing Software Architecture based on Statechart Semantics*. in *15th Brazilian Symposium on Software Engineering (SBES 2001)*. 2001. Rio de Janeiro, Brazil.
10. Booch, G., R. J., and I. Jacobson, *The Unified Modeling Language User Guide*.
11. Xu, L., Dias M., and Richardson D. *Regression Testing via Model Checking*. in *IASTED International Conference on Software Engineering*. Innbruck, Austria.
12. Pavlopulou, C. and Young M.. *Residual Test Coverage Monitoring*. in *21st international conference on Software engineering*. 1999. Los Angeles, California, United States: IEEE Computer Society Press.
13. Naslavsky, L., Dias M., and Richardson D. *Multiply-Deployed Residual Testing at the Object Level*. in *IASTED International Conference on Software Engineering (SE2004)*. 2004. Innsbruck, Austria.
14. *OMG UML Specification v.14*.