Modelling Dynamic Component Dependencies

Siamak Haschemi and Daniel A. Sadilek
Humboldt-Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
{haschemi|sadilek}@informatik.hu-berlin.de

Abstract. This paper is about modelling dynamic dependencies of components as required in dynamic environments. We sketch a formal model for describing the dependencies of software components on hardware and other software components. In a unified way, we represent software components and hardware components with their properties. The properties can be changed during runtime. Expressions over properties define component dependencies, which are steadily resolved to deal with dynamics.

1 Introduction

This paper is concerned with software components that operate in dynamic environments. Such a dynamic environment is, for example, a wireless mesh network where the topology fluctuates and hardware can fail [1]. A software component can depend on hardware and other software components. Consider, for example, a software component that should only run at a certain minimum current of a battery. If such a dependency fails, the software component should be stopped. This may trigger to stop further software components. If the battery gets recharged, the software components should be started again.

Dynamic dependencies like these cannot be explicitly described in the “standard” component models, such as JavaBeans, J2EE, COM, or CCM. Instead, they must be handled programatically. In the OSGi community, there are a couple of frameworks with a declarative language for dynamic dependency description and runtime composition [2,3]. But they are tightly coupled to the OSGi specification.

In this paper, we present a simple, formal, platform independent model for explicitly describing dynamic component dependencies. The model’s purpose is to decide at runtime which software components can be started. We regard as components both software components and hardware components, such as the battery. To describe dependencies, we assign properties to components and use them in dependency expressions. If all dependencies of a component can be resolved, it can be started. The properties can be changed during runtime. Therefore, in an implementation of our model, the dependencies must be continuously resolved.
A = unicode characters
Identifier = [A-Za-z][A-Za-z0-9]*
K = Identifier
V = A* ∪ Z
Mul = Z+ × (Z+ ∪ {∗})
lowerbound : Mul → Z+
upperbound : Mul → Z+ ∪ {∗}
with ∀(a,b) ∈ Mul(b = ∗ ∨ a ≤ b)
Name = Identifier
Exp := Exp BoolBinOp Exp
| ↑ Exp | (‘Exp’)
| ValueExp ValueBinOp ValueExp
BoolBinOp ::= ‘&’ | ‘|’ | ‘¬’
ValueBinOp ::= ‘=’ | ‘<’ | ‘>’ | ‘>=' | ‘<’ | ‘<=’
ThisProp ::= K ‘(‘ this’ ‘)’
OtherProp ::= K ‘(‘ other’ ‘)’ | K

Fig. 1. Mathematical framework for describing components.

Exp ::= Exp BoolBinOp Exp
| ↑ Exp | (‘Exp’)
| ValueExp ValueBinOp ValueExp
BoolBinOp ::= ‘&’ | ‘|’ | ‘¬’
ValueBinOp ::= ‘=’ | ‘<’ | ‘>’ | ‘>=' | ‘<’ | ‘<=’
ThisProp ::= K ‘(‘ this’ ‘)’
OtherProp ::= K ‘(‘ other’ ‘)’ | K

Fig. 2. Grammar for expressions over properties.

2 Modelling and Resolving Component Dependencies

An overview of the formal model is given by Fig. 1.

Component Properties. We denote the set of all components with C. Components have properties consisting of a key and a value. We encode the properties of components in the function φ : C × K → V.

Expressions over Properties. In the following, we will model dependencies and their precedences. For this, we use boolean-valued expressions over properties. For example, to model the “battery dependency” from the introduction, we could write: «name = “battery” & current ≥ 2».

We define the set Exp of possible expressions with the grammar shown in Fig. 2. We assume that the operators have the standard precedence and that string values are enclosed in double quotes. An important part of expressions is the property access. A property access can refer to the component itself or to the component that it depends on. In the first case, we write «prop(this) », in the second case we write «prop(other) » or simply «prop ».
eval(exp₁ \& exp₂, c₁, c₂) := eval(exp₁, c₁, c₂) \& eval(exp₂, c₁, c₂)

(‘|’ and ‘→’ are analog)

\[.eval(exp, c₁, c₂) := \neg eval(exp, c₁, c₂)\]

\[eval(‘exp’), c₁, c₂) := eval(exp, c₁, c₂)\]

\[eval(valueExp₁ = valueExp₂, c₁, c₂) := eval(valueExp₁, c₁, c₂) = eval(valueExp₂, c₁, c₂)\]

(‘>', ‘\geq’, ‘<’, and, ‘\leq’ are analog)

\[eval(v, c₁, c₂) := v\]

\[eval(k (‘this’), c₁, c₂) := \phi(c₁, k)\]

\[eval(k (‘other’), c₁, c₂) := \phi(c₂, k)\]

\[eval(k, c₁, c₂) := \phi(c₂, k)\]

Fig. 3. The function eval for evaluating expressions. Variable names stem from the non-terminal names that the variables range over.

To evaluate expressions, we define two functions eval \(_e\) : Exp \(\times\) C \(\times\) C \(\rightarrow\) \{true, false\} and eval \(_v\) : ValueExp \(\times\) C \(\times\) C \(\rightarrow\) V by structural recursion over the syntax of expressions. They are given by Fig. 3.

**Modelling Component Dependencies.** A component can depend on other components. We model such dependencies with a name, a multiplicity, an expression for the dependency itself, and an expression that determines a precedence. The set of all dependencies is Dep. We assign dependencies to components with the relation \(\delta \subseteq C \times Dep\). In the following, we explain all constituents of a dependency.

**Dependency Expression** (depexp : Dep \(\rightarrow\) Exp): It determines which components can match the dependency.

**Dependency Name** (name : Dep \(\rightarrow\) Name): Using a dependency’s name, a component implementation can access all components that match the dependency.

**Dependency Multiplicity** (mul : Dep \(\rightarrow\) Mul): There may be multiple components that match a dependency. We model this using multiplicities that consist of a lower and an upper bound. The lower bound determines the minimum number of components that must match the dependency expression. The upper bound determines how many matching components can be accessed by the component that has the dependency.

**Dependency Precedence** (preexp : Dep \(\rightarrow\) Exp): If the number of components that match a dependency is greater than some upper bound \(b\) of the dependency’s multiplicity, a hint is necessary which components should be used. We model this hint as a precedence, which we define as a boolean-valued sort predicate over the properties of two components.

**Resolving Component Dependencies.** When all dependencies of a component can be resolved, it can be started; if not, it cannot be started and must be stopped if it has been started before. In an implementation, the dependency
resolution must be performed whenever a property changes or a new component is added.

To decide if a certain dependency can be resolved, we first need all matching components. We define the function \( \text{matches} : C \times \text{Dep} \rightarrow \mathcal{P}(C) \), where \( \mathcal{P}(C) \) denotes the power set of \( C \), with

\[
\text{matches}(c, d) = \{ c' | \text{eval}(\text{delexp}(d), c, c') \}.
\]

To determine if a dependency is resolved, we define the function \( \text{res}_\text{dep} : C \times \text{Dep} \rightarrow \{ \text{true}, \text{false} \} \). If all dependencies of a component are resolved, the component itself is resolved and can be started. To determine the resolved components, we define the function \( \text{res}_\text{com} : C \rightarrow \{ \text{true}, \text{false} \} \). The functions \( \text{res}_\text{dep} \) and \( \text{res}_\text{com} \) depend on each other, because a dependency can only be resolved by components that are themselves resolved:

\[
\text{res}_\text{dep}(c, d) = \# \{ c' | \text{matches}(c', d) \land \text{res}_\text{com}(c') \} \geq \text{lowerbound} \text{mul}(d)
\]
\[
\text{res}_\text{com}(c) = \forall d \in \{ d | (c, d) \in \delta \} (\text{res}_\text{dep}(c, d))
\]

Therefore, these functions must be computed recursively. If a circular dependency is encountered, it is considered resolved.

3 Conclusion

The formal model we presented allows for modelling and dynamically resolving component dependencies. Its key features are dynamic component properties and expressions over them for describing dynamic component dependencies. By describing such dependencies explicitly, our model enables static analysis and reasoning about dependencies.

In this paper, we focused on dynamic dependencies and used expressions over simple integer and string properties to model them. For a feature-complete component description language, our formal model must be extended with concepts for more complex types to describe interfaces. This could be done by aligning it with an existing interface description language, such as Corba IDL or WSDL [4].

References