A Temporal Logic Approach to the Specification of Reconfigurable Component-based Systems[*]

Nazareno Aguirre[†] Tom Maibaum
Department of Computer Science, King's College, Strand, London WC2R 2LS, United Kingdom
aguirre@dcs.kcl.ac.uk, *tom@maibaum.org*

Abstract

We propose a formal specification language for dynamically reconfigurable component-based systems, based on temporal logic. The main aim of the language is to allow one to specify behaviours of component-based systems declaratively, with special emphasis on behaviours in which the architectural structure of the system changes dynamically.

Due to the semantics and organisation of our language, it is straightforward to hierarchically build reconfigurable systems in terms of subsystems and basic component parts, and reason about them within the language. Despite its expressive power, the language is rather simple.

1 Introduction

In component-based systems, the notion of architecture has come to play a key role. Special specification languages, called architecture description languages (ADLs), were proposed to describe and analyse properties of (sometimes evolving) architectures [9]. Many of these are able to deal with what is called dynamic reconfiguration, i.e., with the description of operations which may modify the system's structure at run time. While ADLs provide constructs for modelling the architecture of a system, they often do not support a language for reasoning about possible system evolution. In other words, ADLs support the definition of components, interconnections and transformation rules or operations for making architectures evolve, but any kind of reasoning about behaviours is often performed in some "meta-language". We are interested in specifying and reasoning about the increasingly important problem of dynamic reconfiguration of component based systems. To our knowledge, approaches to the specification of reconfigurable systems are either informal, making reasoning about system properties very hard(!), or are either operational (e.g., graph grammar based mechanisms) [2][11] or chemical abstract machine based [6], in either case forcing the specifier to reason about reconfiguration properties outside the language, in some (informal) meta-language.

We wish, then, to be able to specify and reason about the consequences of using certain reconfiguration operations in a declarative manner, adding abstraction to what, to our understanding, can be operationally specified by ADLs such as the ones described in [2][11]. We choose to use temporal logic as the formal basis for our language.

The language is organised around: the notion of components, which are represented by classes that define templates for these components; the notion of connector type, which we call associations, which are then used to define the potential ways in which components may communicate in a system; the notion of subsystem, the new notion that defines the unit of modularity from which reconfigurable systems are built, and which conveys the information about what components, what associations and what reconfiguration operations are used to define the module. By providing a calculus to support each level of subsystem construction, we are able to reason about individual components, individual associations and about modules built out of such components and associations. In particular, we are able to assert and prove dynamic reconfiguration properties of specified systems.

2 Specifying Classes

Classes are the basic building blocks in the language. They describe templates of the most basic components, since more complex kinds of components can also be defined. In analogy with object-oriented languages, classes in our language are units of modularisation that contain data and behaviour (actions). However, our classes are different from classes in OO languages. The main difference is that we do not consider classes as valid types of class attributes.

[*]This work was partially supported by the Engineering and Physical Sciences Research Council of the UK, Grant Nr. GR/N00814.

[†]On leave from Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina.

Attributes in a class definition must be of basic type, forbidding any kind of references to components inside other components. As usual in some ADLs, communication between components is achieved by means of architectural connectors [2], external to the definition of components.

Assuming that an algebraic specification $ADT = (\Sigma, E)$ for datatypes is given, a *class signature* consists of a finite set of read variables, a finite set of attribute symbols and a finite set of action symbols. Both read variables and attribute symbols have an associated *type*, i.e. a sort in Σ. Action symbols have arguments, each of which has a type from Σ.

A class signature defines the extra-logical symbols used in the vocabulary for the class. Using these symbols and some logical symbols, we build formulae to specify the intention of the actions of the class. A *class specification* then is simply a class signature C and a finite set of temporal formulae over C, called the *axioms* of the specification. For a description on how these formulae are constructed, see [1].

Consider the following class specification:

Class *Printer*

Exports $\mathit{print}(), \mathit{load}(\mathsf{string}), ready$

Attributes

ready : boolean
job : string

Actions

$\mathit{print}()$
$\mathit{load}(\mathsf{string})$
$\mathit{print\text{-}el}(\mathsf{char})$

Axioms

1. $\mathbf{BEG} \to \mathit{job} = []$
2. $\forall s \in \mathsf{string} : \mathit{load}(s) \to \mathit{job} = []$
3. $\forall s \in \mathsf{string} : \mathit{load}(s) \to \bigcirc(\mathit{job} = s)$
4. $\mathit{print}() \to \mathit{job} \neq []$
5. $\forall j \in \mathsf{string} : \mathit{print}() \wedge \mathit{job} = j \to \mathit{print\text{-}el}(\mathit{head}(j))$
6. $\forall j \in \mathsf{string} : \mathit{print}() \wedge \mathit{job} = j \to \bigcirc(\mathit{job} = \mathit{tail}(j))$
7. $\mathit{job} \neq [] \to \Diamond(\mathit{print}())$
8. $\mathit{ready} = \mathsf{T} \leftrightarrow \mathit{job} = []$

EndofClass

Axioms indicate the intention of the actions of the class, i.e., their effect on attributes. Since the extra-logical symbols used for these axioms are based on the signature of the class, a "linguistic" locality is enforced: nothing "outside" the class can be mentioned. The intuitive use of the temporal operators to provide meaning to actions is easy to understand. For instance, the first axiom states that in the initial state of an instance, denoted by **BEG**, the attribute *job* is empty; the second one says that the action *load* can only take place when there is no job already in the printer; the third one says that if $\mathit{load}(s)$ takes place, then s becomes the current job of the instance (in the next state).

Read variables are a special kind of attribute. They are used by the components to get information from the environment, in the same way input variables are used in CommUnity [5]. Consider the following class specification:

Class *Server*

Exports $\mathit{enqueue}(\mathsf{string}), \mathit{print}()$

Read Variables

$\mathit{p\text{-}ready}$: boolean

Attributes

$\mathit{p\text{-}queue} : \mathit{list}(\mathsf{string})$

Actions

$\mathit{enqueue}(\mathsf{string})$
$\mathit{print}()$
$\mathit{send}(\mathsf{string})$

Axioms

1. $\mathbf{BEG} \to \mathit{p\text{-}queue} = []$
2. $\forall s \in \mathsf{string}\, \forall q \in \mathit{list}(\mathsf{string})$:
$$\mathit{enqueue}(s) \wedge \mathit{p\text{-}queue} = q \to \bigcirc(\mathit{p\text{-}queue} = q +\!\!\!+ s)$$
3. $\mathit{print}() \to \mathit{p\text{-}queue} \neq []$
4. $\forall q \in \mathit{list}(\mathsf{string}) : \mathit{print}() \wedge \mathit{p\text{-}queue} = q \to$
$$[\mathit{send}(\mathit{head}(q)) \wedge \bigcirc(\mathit{p\text{-}queue} = \mathit{tail}(q))]$$
5. $\mathit{p\text{-}queue} \neq [] \to \Diamond(\mathit{print}())$
6. $\forall s \in \mathsf{string} : \mathit{send}(s) \to \mathit{p\text{-}ready} = \mathsf{T}$
7. $\forall s \in \mathsf{string} : \mathit{send}(s) \to \mathit{print}()$

EndofClass

Server is the specification of a print server, with a printing queue and basic operations. The read variable *p-ready* tells the server whether the system is ready to print or not (see Axiom 6). From the point of view of the component, read variables belong to the "outside world", and therefore they can change their state arbitrarily.

2.1 Proving Properties of Classes

We use as a basis for our logic the one given by Manna and Pnueli for reactive systems [7]. One of the extensions to it is that we consider a *starting point* in time, denoted, as shown previously, by **BEG**. Also, the reasoning concerning datatypes is performed outside this logic, using equational logic from algebraic specifications. This could be considered another level of the language, lower than that of class specifications. The standard inference rules are extended, in order to be able to import datatypes properties into the logic.

The way properties are expressed using temporal logic is clear from the axioms of the above specifications. In fact, the use of temporal logic as a specification language of state-based systems is well-known. The theory obtained by closing the axioms of a class C, the explicit and some implicit ones defined in [1], under the inference rules represents the set of all (expressible) provable properties of the instances of C. Since classes denote standard temporal logic theories, we can give a model-theoretic semantics by simply interpreting these theories in the standard way, using Kripke structures.

3 Specifying Associations

Once classes are described, interactions between classes can be defined. The interaction between class instances is achieved via action synchronisation and attribute sharing, in the style used in [3][4]. The syntax of associations is simple— we only need to state which actions must synchronise, and which attributes or read variables in the participants are identified. An *association* then consists of a finite (and nonempty) set of participating identifiers, each of which has an associated class name, and a finite set of synchronisation definitions, of the form $x.a \leftrightsquigarrow y.a'$, where x, y are participants of classes C_x and C_y respectively; a, a' are either attributes of C_x and C_y respectively, or action symbols of the corresponding classes. Further restrictions on synchronisations are defined in [1].

Consider the following association between printers and servers:

Association USES

Participants

$SRV : Server$
$PR : Printer$

Connections

$SRV.p\text{-}ready \leftrightsquigarrow PR.ready$
$\forall j \in \mathsf{string} : PR.load(j) \leftrightsquigarrow SRV.send(j)$

EndofAssoc

Associations are like *templates* of connectors [2], in the same way that classes are considered *templates* of components in our language. They are characterised by formulae in the next layer of the language, the subsystems layer.

4 Specifying Subsystems

A subsystem is a new unit of modularisation in our language, which is needed to put together associations and classes. Each subsystem specifies which instances of each of the classes are present in it, and how they are related, via association instances. Operations that modify the state of the subsystem can be defined at this stage. These changes can be either creation or deletion of instances, or changes in the architectural configuration of it, that is, relating unrelated components, or disassociating related ones. Consider the following subsystem specification, where we put together servers and printers:

Subsystem Multiple_Printers

Initial State

$S : Server$
$P_1 : Printer$
$USES(S, P_1)$

Operations

$change(y : Printer)$
$add(y : Printer)$

Axioms

1. $\forall s, p_1, p_2 : USES(s, p_1) \wedge USES(s, p_2) \rightarrow p_1 = p_2$
2. $\forall p : change(p) \rightarrow \bigcirc(USES(S, p))$
3. $\forall p : \neg change(p) \rightarrow [\forall x, y : USES(x, y) \leftrightarrow \bigcirc USES(x, y)]$
4. $\forall p : change(p) \rightarrow (S.p\text{-}ready = \mathsf{T})$
5. $\forall p : add(p) \leftrightarrow [(\neg Printer(p)) \wedge \bigcirc Printer(p)]$
6. $\forall s : Server(s) \leftrightarrow \bigcirc Server(x)$
7. $\forall p : Printer(p) \rightarrow \bigcirc(Printer(p))$

EndofSubsystem

Axioms in subsystems are used in the same way they are used in classes. For instance, the first axiom states an *integrity constraint* for association *USES*, which specifies that a server can be using at most one printer at a time; Axiom 2 says that the occurrence of *change*(p) makes server S to use p.

4.1 Some Properties of Subsystems

There exist several properties of subsystems that are not explicitly specified in a specification, but instead are characterised by a set of automatically generated axioms, to be included in the theory associated with the subsystem. As can be seen from the axioms of the previous subsystem, special predicates (see axioms 5 and 6 in the subsystem) are used to characterise the live instances. In our case, predicates *Server*(x) and *Printer*(y) indicate that x is a live server and y a live printer, respectively. Some of the generated axioms characterise properties of these predicates. Axioms regarding the typing conditions of associations and actions are also given. For our association *USES*, we have the following:

$$\forall x, y : USES(x, y) \rightarrow Server(x) \wedge Printer(y)$$

The "initial state" declaration indicates which instances are considered live when the subsystem is created. Special formulae are also incorporated in the theory of a subsystem to give meaning to this clause. For a description of the formulae necessary for this and other characterisations see [1].

Associations are also interpreted as automatically generated temporal formulae in the language of subsystems. For our association *USES*, the corresponding formulae to include in any subsystem using it are the following:

$$\forall x, y : USES(x, y) \rightarrow (\forall j \in \mathsf{string} : x.send(j) \leftrightarrow y.load(j))$$
$$\forall x, y : USES(x, y) \rightarrow (x.p\text{-}ready = y.ready)$$

A component may be associated to many other components via associations. In a case where several objects are related to certain object A via an association R, we assume that the synchronised actions (with respect to R) in the client objects cannot occur at the same time. In other words, A has a built-in mutual exclusion mechanism with respect to actions synchronised by R. Again, this is expressed by means of further temporal formulae [1].

4.2 Proving Properties of Subsystems

The proof calculus for subsystems is similar to the one for class specifications. The only difference is that we provide an extra inference rule, which allows us to promote properties from the classes into the subsystem, as properties of the instances.

> If C is a class definition, and $\vdash_A P$, then in any subsystem Sub, the following holds:
>
> $$\forall x : C(x) \rightarrow x.P$$
>
> where $x.P$ denotes the translation of formula P to the language of the subsystem Sub (recall that all formulae coming from classes are *relativised* to the corresponding instance when considered in a subsystem).

Intuitively, this inference rule says that, if x is a live instance of class C, then it has all the properties that can be proved in class C.

For instance, this rule allows us to promote axiom 6 from *Server* to the following property in Multiple_Printers:

$$\forall x : \mathit{Server}(s) \rightarrow (\forall s \in \mathsf{string} : x.\mathit{send}(s) \rightarrow x.\mathit{p\text{-}ready} = \mathsf{T})$$

4.2.1 Sample Properties

Temporal logic provides an expressive and declarative language to state properties. Consider for example the following properties, which were proved to be consequences of the previous subsystem definition:

Property 1: "During the lifetime of a Multiple_Printers subsystem, S is the only server". It can be stated easily:

$$\forall x : \mathit{Server}(x) \rightarrow (x = S)$$

Property 2: "If a printer is being used by a server, then it continues to be used by the same server until the current print job is finished". It is expressed using association *USES*, as follows:

$$\forall x, y : \mathit{USES}(x, y) \rightarrow [\mathit{USES}(x, y)\mathcal{U}(y.\mathit{job} = [\,])]$$

5 Conclusions

We have presented a language for declarative specification and reasoning of component-based systems. The way in which associations are represented in the language, standard in ADLs [8][9], allows it to express properties concerning the architecture of the system in a declarative way. Hence, operations that may change the topology of the system can be easily specified in the formalism. The semantics of the language is defined in terms of a logic based on the combination of first-order and temporal logics. Their proof calculi can be used to prove properties of the system, including dynamic reconfiguration properties. Our formalism could be used to provide both semantics and proof mechanisms for ADLs.

Our work is based on the logical and semantic foundations of [3][4], and therefore our formalism is closely related to the one described in [10][11], for dynamic architecture reconfiguration. That work is based on graph grammars, to provide semantics for reconfiguration, as opposed to ours, almost completely based on temporal logic. We think our work complements the one in [10][11] and other ADLs [2][6], by providing a uniform language to state and prove properties, that could then be related to specifications in ADLs.

References

[1] N. Aguirre and T. Maibaum, *Reasoning about Reconfigurable Object-Based Systems in a Temporal Logic Setting*, in Proceedings of IDPT 2002.

[2] R. Allen and D. Garlan, *Formalizing Architectural Connection*, in Proceedings ICSE '94, Sorrento, Italy, 1994.

[3] J. Fiadeiro and T. Maibaum, *Temporal Theories as Modularisation Units for Concurrent System Specification*. Formal Aspects of Computing, vol. 4, No. 3, Springer-Verlag, 1992.

[4] J. Fiadeiro and T. Maibaum, *Design Structures for Object-Based Systems*. In Formal Methods and Object Technology, S. Goldsack and S. Kent (eds), Springer-Verlag, 1996.

[5] J. Fiadeiro and T. Maibaum, *Categorical Semantics of Parallel Program Design*, Science of Computer Programming 28(2-3), 1997.

[6] P. Inverardi and A. Wolf, *Formal Specification and Analysis of Software Architetures using the Chemical Abstract Machine*, IEEE Transactions in Software Engineering, 1995.

[7] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.

[8] N. Medvidovic, *ADLs and Dynamic Architecture Changes*, in Proceedings of the Second Int. Software Architecture Workshop (ISAW-2), 1996.

[9] N. Medvidovic and R. Taylor, *A Framework for Classifying and Comparing Architecture Description Languages*, In ESEC-FSE'97, 1997.

[10] M. Wermelinger and J. Fiadeiro, *Algebraic Software Architecture Reconfiguration*, in ESEC/FSE'99, LNCS 1687, Springer-Verlag, 1999.

[11] M. Wermelinger, A. Lopes and J. Fiadeiro, *A Graph Based Architectural (Re)configuration Language*, in ESEC/FSE'01, V.Gruhn (ed), ACM Press, 2001.