Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 108 (2004) 69–81

www.elsevier.com/locate/entcs

Hierarchical Temporal Specifications of Dynamically Reconfigurable Component Based Systems

Nazareno Aguirre[a,b,1] Tom Maibaum[b,1]

[a] *Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Enlace rutas 8 y 36 Km. 601, Río Cuarto (5800), Córdoba, Argentina*

[b] *Department of Computer Science, King's College London, Strand, London WC2R 2LS, United Kingdom*

Abstract

We study how temporal specifications of reconfigurable component based systems can be hierarchically organised. We do so by extending a previously introduced declarative prototypical language to admit the definition of hierarchical subsystems. Each subsystem has an internal architecture, composed of its internal interacting (simpler) subsystems, and basic components. The internal architecture of a subsystem can change at "run time" by means of reconfiguration operations.
The notion of subsystem provides an extra coarse grained unit of modularisation, that complements that of components. Since component interaction is achieved by means of *coordination*, a component or subsystem can be represented by a logical theory isolated from the rest of the system. This, in combination with the possibility of hierarchically organising a specification, has a special impact in reasoning, since it allows us to further localise the proof efforts to the relevant subparts of a specification.

Keywords: Software architectures, dynamic reconfiguration, temporal logic

1 Introduction

Software architectures can be regarded as a branch of software engineering that puts emphasis on the high level structure of systems [9,7]. An *architecture* of a system is described in terms of *components*, which can interact

[1] Authors' email addresses: `naguirre@dc.exa.unrc.edu.ar`, `tom@dcs.kcl.ac.uk`.

doi:10.1016/j.entcs.2004.01.013

via defined *connectors*. Connectors are the means for communication between components, and have the particular property of being less cohesive than other communication mechanisms, since they are external to the definitions of components.

A topic that gained attention in the past few years is the possibility of changing the architecture of a system at run time, a feature often called *dynamic reconfiguration* [13]. The need for dynamic reconfiguration appears naturally and frequently, perhaps due to the wide acceptance of *object oriented* modelling and programming [14], where dynamic reconfiguration is straightforwardly supported.

Several *architecture description languages* (ADLs) support dynamic reconfiguration, through the definition of reconfiguration operations. However, those ADLs with support for reconfiguration and formal semantics generally allow only for operational (as opposed to declarative) descriptions. With this restriction in mind, we proposed a declarative formalism based on temporal logic, for the description of reconfigurable software architectures [1]. The abstraction gained by using a declarative framework allows us to study possible more sophisticated abstract ways of describing software architectures. The main characteristics of this formalism are: *(i)* it has direct support for reasoning, due to its logical nature, *(ii)* it is expressive enough to allow for the description of components in a property oriented way, and *(iii)* components and configurations are uniformly represented by logical theories, which allows us to build hierarchical organisations of systems [2].

Configurations of components are encapsulated into *subsystems*, which can be dynamically reconfigured via subsystem operations. We have already justified the technical possibility of hierarchically organising reconfigurable systems in terms of subsystems and components in [2]. However, the exact difficulties and the advantages of doing so remained to be studied. This paper argues about some important advantages of hierarchical organisations of component based systems, by using the notion of subsystem, and proposes an extension to a previously introduced prototypical specification language to cope with hierarchical subsystems.

2 A Temporal Specification Language

In this section we describe a language over which the study of hierarchical subsystems is based. The language is prototypical, and should not be regarded as a real ADL. It is intended to be just a means to study more abstract and declarative ways of describing software architectures, and to probe the capabilities of our proposed formalism.

The language is a simple front end to a temporal logic. The main characteristics of this logic are: *(i)* it is first-order, *(ii)* time is linear, discrete and with an initial instant of time (i.e., the model of time is $\mathbb{N}$), *(iii)* besides the usual connectives and quantifiers, the logic also features the temporal operators $\bigcirc$ ("*next*"), $\Box$ ("*always in the future*"), $\Diamond$ ("*eventually in the future*") and $\mathcal{U}$ ("*strong until*"), *(iv)* some function and predicate symbols (called *flexible*) are interpreted in a state dependent way, although there also exist functions and predicates with state independent interpretations (called *rigid*).

This logic is a variant of the Manna-Pnueli logic [12], in which the flexible symbols, i.e., those whose interpretation is state dependent, have been generalised. Unfortunately, due to space restrictions, we are unable to discuss more details regarding this logic. We refer the interested reader to [12] for details on the original Manna-Pnueli logic, and to [2] for a detailed description of our variant of the logic.

The logic, with a certain kind of language translation, constitutes a π-institution [6,10]. This implies a useful structural property, that enables us to *promote* properties from the lower layers of a system to their including subsystems [2].

We summarise below the main constructs of the language and its properties. The style of specification is inspired by [5] and related work. Particularly, we follow several ideas put forward with the CommUnity design language [15].

2.1 Describing Components

The lowest layer of the language is composed of a specification $\mathcal{ADT}$ of datatypes. It is simply a theory presentation over an alphabet without flexible symbols (equivalent to a first order logic characterisation of datatypes).

Let $\mathbf{S}_{\mathcal{ADT}}$ be the set of sorts of the datatype specification $\mathcal{ADT}$; we can define a component signature by providing: *(i)* a set of $\mathbf{S}_{\mathcal{ADT}}$-indexed read variables, *(ii)* a set of $\mathbf{S}_{\mathcal{ADT}}$-indexed attributes, and *(iii)* a set of $\mathbf{S}^*_{\mathcal{ADT}}$-indexed actions. The *state* of a component is determined by its attributes, which are like variables of imperative programming languages. Read variables are simply special attributes, out of the control of the component and used as "entry points", for implementing communication. Actions represent parameterised instantaneous operations of the component.

The intended behaviour of components is described by temporal axioms, employing: *(i)* datatypes specified in $\mathcal{ADT}$, *(ii)* read variables and attributes as flexible 0-ary function symbols, *(iii)* actions as flexible predicate symbols, and *(iv)* a special 0-ary flexible predicate, which denotes the activeness of the component. The combination of a signature for components together with axioms to characterise the components is actually the description of a component type, that we call *class definition*. Class definitions are given a name, which is also used as the special 0-ary flexible predicate denoting the component's activeness.

Possible counterparts of class definitions in some ADLs are *component types* in Darwin [11], *component definitions* in Acme [8] and in Wright (within styles) [3], and *programs* in CommUnity [15].

Example 2.1 Let us provide an example of a class definition. Suppose we want to specify a network of "units" that can interchange messages. Let us call these units *cells*. Cells are associated an address, which is an integer number, and is supposed to be unique to a cell. Messages include the address of the destination cell.

We can model messages and addresses as basic datatypes. So, let us assume that our datatype specification $\mathcal{ADT}$ contains, besides the specification of the usual datatypes such as booleans, strings, integers, etc, the specification of a special datatype $\mathsf{message}$. There exists also a (static) function $dest : \mathsf{message} \rightarrow \mathsf{integer}$, which singles out the destination address embedded in a message. There is a special "empty" message, denoted by the 0-ary function symbol $null :\rightarrow \mathsf{message}$. The destination address of *null* is undefined.

We can specify cell components by defining a class, as shown in Fig. 1. Class *Cell* has two boolean read variables, *in* and *out*, which indicate a cell whether there is an incoming message or if the "environment" is ready to receive an outgoing message from the cell, respectively. It also has an integer typed attribute, meant to hold the address of the cell, and two attributes *curr-in* and *curr-out* of type $\mathsf{message}$, which serve the purpose of storing a just received message (ready to be "consumed") and a message ready to be sent, respectively.

The activeness of the component is represented, as we previously indicated, by the flexible predicate named after the class name, i.e., predicate *Cell*. Intuitively, the truth of predicate *Cell* at an instant of time should be interpreted as the component being active, or "live" during that instant. By contrast, if predicate *Cell* is not true at an instant of time, then the component is not active, or "dead", at that instant.

Class *Cell* contains six (instantaneous) actions. Action *c-init*($\mathsf{integer}$) is an initialisation operation, which sets the *address* attribute (see Axiom 1). As expressed by Axiom 2, *c-init* can be called once per life time of a cell component. The intuitive reading of Axiom 2 is: "in all states and for all $x \in \mathsf{integer}$, it is the case that, if the instance is active and *c-init(x)* occurs, then it will not occur again in the current life time of the instance" [2]. Axiom

[2] Axiom 2 employs a derived temporal operator, namely the $\mathcal{W}$ operator ("weak until").

Class *Cell*
Read Variables in, out : boolean
Attributes $address$: integer; $curr\text{-}in, curr\text{-}out$: message
Actions $c\text{-}init$(integer), $prod$(message), $send$(message), get(message), $cons$(message), $rem()$
Axioms
1. $\Box[\forall x \in \text{integer} : Cell \wedge c\text{-}init(x) \rightarrow \bigcirc(address = x)]$
2. $\Box[\forall x \in \text{integer} : Cell \wedge c\text{-}init(x) \rightarrow \bigcirc(\neg \exists y \in \text{integer} : c\text{-}init(y) \mathcal{W} \neg Cell)]$
3. $\Box[Cell \wedge (address \neq \bigcirc address) \rightarrow \exists x \in \text{integer} : c\text{-}init(x)]$
4. $\Box[\forall m \in \text{message} : Cell \wedge get(m) \rightarrow ((in = \mathsf{T}) \wedge (curr\text{-}in = null))]$
5. $\Box[\forall m \in \text{message} : Cell \wedge get(m) \rightarrow \bigcirc(curr\text{-}in = m)]$
6. $\Box[\forall m \in \text{message} : Cell \wedge cons(m) \rightarrow$
$((curr\text{-}in \neq null) \wedge (curr\text{-}in = m) \wedge (dest(m) = address))]$
7. $\Box[\forall m \in \text{message} : Cell \wedge cons(m) \rightarrow \bigcirc(curr\text{-}in = null)]$
8. $\Box[Cell \wedge rem() \rightarrow ((curr\text{-}in \neq null) \wedge (dest(curr\text{-}in) \neq address))]$
9. $\Box[Cell \wedge rem() \rightarrow \bigcirc(curr\text{-}in = null)]$
10. $\Box[Cell \wedge (curr\text{-}in \neq \bigcirc curr\text{-}in) \rightarrow (\exists m \in \text{message} : get(m) \vee cons(m)) \vee rem()]$
11. $\Box[\forall m \in \text{message} : Cell \wedge send(m) \rightarrow ((out = \mathsf{T}) \wedge (curr\text{-}out = m))]$
12. $\Box[\forall m \in \text{message} : Cell \wedge send(m) \rightarrow \bigcirc(curr\text{-}out = null)]$
13. $\Box[\forall m \in \text{message} : Cell \wedge prod(m) \rightarrow (curr\text{-}out = null)]$
14. $\Box[\forall m \in \text{message} : Cell \wedge prod(m) \rightarrow \bigcirc(curr\text{-}out = m)]$
15. $\Box[Cell \wedge (curr\text{-}out \neq \bigcirc curr\text{-}out) \rightarrow \exists m \in \text{message} : send(m) \vee prod(m)]$
EndOfClass

Fig. 1. Class *Cell*: A Basic Class Definition

3 indicates that, while the component is active, attribute *address* can only be modified by action *c-init*.

Action *get* obtains an incoming message from the environment. It has as a precondition that *in* must be true (there is a message waiting to be obtained) and *curr-in* must be *null* (no incoming messages are overwritten before being consumed) (see Axiom 4). After $get(m)$ occurs, *curr-in* becomes m in the next state (see Axiom 5). Action *cons* consumes a previously obtained message, provided that the incoming message is addressed to the component (see Axioms 6-7). If a previously obtained message is not addressed to the component, it can be removed using the $rem()$ operation (see Axioms 8-9). Actions *prod* and *send* are meant to produce and send messages, respectively. They are characterised by Axioms 11-14.

From the previous example, the reader might get an idea of the way that some temporal operators are used to specify the intention of actions. In particular, the example illustrates the use of the temporal operators $\Box$ (always in the future), to represent invariant properties of components (as used in all axioms), and $\bigcirc$ (next) to specify postconditions of actions, as in Axioms 1,5,7.

The use of the predicate *Cell* (the flexible predicate named after the class name) in the axioms might appear to be a bit unnatural at a first glance. It is necessary to understand that axioms of components are absolute, in the sense that they speak about all the possible states of a system in which the component is engaged, including those in which the component is not live. In addition, the availability of such predicates in the language of components allows the specifier to enforce certain constraints. For instance, the constraint "a component cannot be 'unplugged' while it is engaged in certain activity", can be specified *within* a component's theory, precisely due to the fact that this kind of predicate is available in the vocabulary of components.

A class specification C is interpreted as a temporal theory presentation. The axioms of the presentation are obtained by putting together: *(i)* the axioms explicitly provided as part of the class definition C, *(ii)* the axioms given for the datatype specification $\mathcal{ADT}$, and *(iii)* a special implicit "frame axiom", called *locality axiom*, and expressing that a component must change its state only by means of its defined actions. The theorems of this theory, obtained by means of a proof calculus presented in [2], represent the properties of all "instances" of C.

The monotonicity of the logic and the inclusion of the axioms of $\mathcal{ADT}$ in the theory of a class allow us to reason about datatype properties within $\mathcal{ADT}$ and then "import" these properties in proofs of properties within a class C [2]. For our *Cell* class, a sample theorem of the corresponding theory is the following: "in all states and while the cell is active, it is the case that, actions *rem* and *cons* cannot occur simultaneously". This property, which can be easily proved using the mentioned proof calculus, is represented in the logic by the formula:

$$\Box[\mathit{Cell} \rightarrow \neg(\mathit{rem}() \wedge (\exists m \in \mathsf{message} : \mathit{cons}(m)))].$$

2.2 Describing Interactions

We choose to define class definitions as *closed* independent units. That is, we do not allow classes to refer to other classes within their definitions. This is an important point, since from a logical point of view it allows us to reason about component properties independently of the rest of the system. But, of course, we need ways of making components interact. We achieve communication between components by using the useful concept of *coordination*. In this respect, our means for communication are very close to those of CommUnity [15], although we allow for more flexibility. To make components interact we define *associations*. Associations are composed of a set of participants and a number of formulae, which characterise the interaction.

Consider the association definition of Fig. 2. This association defines a way to make cells communicate. It has two cell typed participants, s and t

(source and target). The connections indicate that, if two cells are connected via *Link*, then; *(i)* the *in* read variable of the first is true iff the *curr-out* attribute of the second is not *null*, and vice versa, *(ii)* the *out* read variable of the first is true iff the *curr-in* attribute of the second is *null*, and vice versa, and *(iii)* the *send* action of the first "calls" the *get* action of the second, and vice versa.

Association *Link*
Participants $s, t : Cell$
Connections
1. $(s.in = \mathsf{T}) \leftrightarrow (t.curr\text{-}out \neq null)$
2. $(s.out = \mathsf{T}) \leftrightarrow (t.curr\text{-}in = null)$
3. $(t.in = \mathsf{T}) \leftrightarrow (s.curr\text{-}out \neq null)$
4. $(t.out = \mathsf{T}) \leftrightarrow (s.curr\text{-}in = null)$
5. $\forall m \in \mathsf{message} : (s.send(m) \rightarrow t.get(m))$
6. $\forall m \in \mathsf{message} : (t.send(m) \rightarrow s.get(m))$

EndOfAssociation

Fig. 2. Association *Link*: A Simple Association.

Note that the formulae that define the connections of *Link* have the participants as free variables. These formulae will be systematically transformed, and will form part of a theory characterising dynamic configurations of components.

3 A Notation for Subsystems

In the previous section, we showed how component and connector types can be declaratively defined, by means of classes and associations, respectively. We need now to compose these specifications in order to build *architectures* of interacting components. We proposed to do so by defining what we called *subsystems* [1]. A subsystem is a new unit of modularisation, which encapsulates a dynamically reconfigurable set of interacting components. Classes are templates of components whose internal structure is basic, composed simply of their attributes; subsystems on the other hand, or more precisely, subsystem instances, can have a complex internal structure, composed of their internal live components and their interconnections. Moreover, subsystems admit the definition of *reconfiguration operations*, which can dynamically change their internal structure.

In a previous work, we defined subsystems as complex components whose internal state is *dynamic*, and is built out of instances of classes (i.e., components) related by means of instances of associations (i.e., connectors) [1]. We want now to extend that, to allow subsystems not only to be composed of instances of classes, but also to subsume instances of simpler subsystems, thus allowing for hierarchical organisations of systems. As a result, we would have two types of component definitions: classes, which define simple unstructured components (i.e., the base case of the process of defining aggregations), and subsystems, which define complex components whose structure is built out of instances of simpler components. This is feasible thanks to the fact that the semantics of basic components and aggregations are defined in a similar way, by means of temporal theories, and therefore there are no technical restrictions for iterating the process of defining aggregations [2].

Let us then propose an extended notation for subsystems. Let $\mathcal{ADT}_{\mathsf{NAME}}$ be a conservative extension of $\mathcal{ADT}$ with an extra sort NAME and a sufficiently large set of constants of this sort. Constants of sort NAME are used to represent identifiers of components of the lower layers. A subsystem signature is composed of: *(i)* a name, *(ii)* finite sets of basic attributes and basic read variables, typed by a sort of $\mathcal{ADT}_{\mathsf{NAME}}$, *(iii)* a finite set of operations, whose arguments are typed by sorts of $\mathcal{ADT}_{\mathsf{NAME}}$.

Attributes are part of the internal state of a subsystem. Read variables will serve the purpose of allowing a subsystem to communicate with others. The operations allow a subsystem to evolve. Contrary to the use of operations in basic components, operations in subsystems can modify the architectural structure of the subsystem, by creating or deleting instances of components, and creating or deleting connections between them. Therefore, we can consider the operations of a subsystem as *reconfiguration* operations, that will change the structural aspect of the subsystem at run time.

In order to logically characterise this, a subsystem Sub is equipped with a finite set of axioms, which are formulae over the alphabet $\mathcal{A}_{Sub}$, composed of: *(i)* the extended datatype specification $\mathcal{ADT}_{\mathsf{NAME}}$, *(ii)* the flexible function and predicate symbols resulting from class definitions **and other more basic subsystems**, adding to all of them an extra parameter of sort NAME, *(iii)* a flexible predicate symbol $R : \underbrace{\mathsf{NAME}, \ldots, \mathsf{NAME}}_{k \text{ times}}$ for each association definition R with k participants, and *(iv)* a flexible predicate symbol $a : S_1, \ldots, S_k$ for each subsystem action of type $a(x_1 : S_1, \ldots, x_k : S_k)$.

We require the sets of symbols originating in class definitions to be disjoint, in order to univocally determine the class a symbol belongs to. This is just to make the presentation simpler.

Example 3.1 With *Cell* and *Link* already defined, we can think of a basic subsystem, *SubNet*, to represent a dynamic aggregation of cells. More precisely, a subnet is a dynamic collection of cells, where there exists a special cell called *gateway*. All other cells within a subnet are connected to the *gateway*, in a “star” topology. New cells can be created, and existing ones deleted, via corresponding operations. The subsystem of Fig. 3 is a possible specification of subnets. Axioms 1-3 specify that, while the subsystem is active, or live, the cells are arranged in a star topology, with *gateway* as the center. Axioms 4-5 express that *gateway* is a cell that does not change during the life time of the subnet. Axiom 6 corresponds to the informal requirement that an address must be unique to a cell[3]. Axioms 7-8 relate the initialisation operation of the subsystem, namely *s-init*, to the creation and initialisation of the *gateway*. Finally, Axioms 9-11 and 12-14 characterise the operations *add-cell* and *rem-cell*, for the creation and deletion of cells within a subnet.

Subsystem *SubNet*
Attributes *gateway* : NAME
Actions *s-init*(integer), *add-cell*(NAME), *rem-cell*(NAME)
Axioms
1. $\Box[\forall n, m : \mathit{SubNet} \wedge \mathit{Link}(n, m) \rightarrow (n = \mathit{gateway})]$
2. $\Box[\forall n : \mathit{SubNet} \wedge \mathit{Cell}(n) \wedge (n \neq \mathit{gateway}) \rightarrow \mathit{Link}(\mathit{gateway}, n)]$
3. $\Box[\mathit{SubNet} \rightarrow \forall n : \neg \mathit{Link}(n, n)]$
4. $\Box[\mathit{SubNet} \rightarrow \mathit{Cell}(\mathit{gateway})]$
5. $\Box[\forall n : \mathit{SubNet} \wedge (\mathit{gateway} = n) \rightarrow ((\mathit{gateway} = n)\mathcal{W}\neg \mathit{SubNet})]$
6. $\Box[\forall n, m : \mathit{SubNet} \wedge \mathit{Cell}(n) \wedge \mathit{Cell}(m) \wedge (n \neq m) \rightarrow (n.\mathit{address} \neq m.\mathit{address})]$
7. $\Box[\forall x \in \mathsf{integer} : \mathit{SubNet} \wedge \mathit{s\text{-}init}(x) \rightarrow \mathit{gateway}.\mathit{c\text{-}init}(x)]$
8. $\Box[\forall x \in \mathsf{integer} : \mathit{SubNet} \wedge \mathit{s\text{-}init}(x) \rightarrow \bigcirc(\forall n : \mathit{Cell}(n) \rightarrow (n = \mathit{gateway}))]$
9. $\Box[\forall n : \forall x \in \mathsf{integer} : \mathit{SubNet} \wedge \mathit{add\text{-}cell}(n, x) \rightarrow \neg \mathit{Cell}(n)]$
10. $\Box[\forall n : \forall x \in \mathsf{integer} : \mathit{SubNet} \wedge \mathit{add\text{-}cell}(n, x) \rightarrow \bigcirc(\mathit{Cell}(n) \wedge n.\mathit{c\text{-}init}(x))]$
11. $\Box[\forall n : \mathit{SubNet} \wedge \neg \mathit{Cell}(n) \wedge \bigcirc(\mathit{Cell}(n)) \rightarrow \exists x \in \mathsf{integer} : \mathit{add\text{-}cell}(n, x)]$
12. $\Box[\forall n : \mathit{SubNet} \wedge \mathit{rem\text{-}cell}(n) \rightarrow \mathit{Cell}(n)]$
13. $\Box[\forall n : \mathit{SubNet} \wedge \mathit{rem\text{-}cell}(n) \rightarrow \bigcirc(\neg \mathit{Cell}(n))]$
14. $\Box[\forall n : \mathit{SubNet} \wedge \mathit{Cell}(n) \wedge \bigcirc(\neg \mathit{Cell}(n)) \rightarrow \mathit{rem\text{-}cell}(n)]$

EndOfSubsystem

Fig. 3. Subsystem *SubNet*: A Basic Subsystem Specification

Subsystems are interpreted as temporal theories, in the same way classes are. The temporal theory for a subsystem *Sub* is constructed from: *(i)* the axioms explicitly provided as part of the *Sub* specification, *(ii)* the axioms in $\mathcal{ADT}_{\mathsf{NAME}}$, and *(iii)* the axioms of the classes **or subsystems** of the lower layer, *relativised* by universally quantifying the extra argument of sort NAME added to read variables, attributes and actions. Indeed, note that some of the axioms of the *SubNet* subsystem use the language of the classes (i.e., of the components of the lower layer). For instance, Axioms 6 and 10 use the "dot notation" (borrowed from object orientation) to denote the "instances" to which attributes or actions correspond to (see, for instance, the expression

[3] Note that we could not express such a requirement within *Cell*'s theory, since it is a *structural* property, and a class's language allows us to refer only to the internal constituents of that class.

$n.address$ in Axiom 6). These prefixes are actually a convenient way of denoting the extra NAME-typed argument incorporated in the read variables, attributes and actions of the class and subsystem definitions of the lower layers. That is, expressions such as $n.address$ and $n.c\text{-}init(x)$ actually correspond to $address(n)$ and $c\text{-}init(x, n)$, respectively [1].

The process of *relativisation* has a nice property: if α is the consequence of a set of formulae Γ, then the relativisation α' of α is a consequence of the relativisation Γ' of Γ [2]. An important consequence of this property is that, since we incorporate the relativisation of the axioms of the lower layers definitions into the including subsystem, all *theorems* (i.e., properties) of the lower layer components are *promoted* into the including subsystem. To clarify this situation, consider for example the following formula:

$$\Box[\forall m \in \mathsf{message} : Cell \wedge cons(m) \rightarrow \bigcirc(curr\text{-}in = null)]$$

This is a trivial theorem within *Cell* (it is, actually, an axiom), whose intuitive reading is: "In all states it is the case that, if the cell is active and $cons(m)$ occurs for some message m, then the attribute *curr-in* is set to *null* in the next state of the system". This property is *promoted* into *SubNet* as the formula:

$$\forall n : \Box[\forall m \in \mathsf{message} : Cell(n) \wedge n.cons(m) \rightarrow \bigcirc(n.curr\text{-}in = null)]$$

whose intuitive reading is: "In all states it is the case that, if n is a live cell and $n.cons(m)$ occurs for some message m, then the attribute *curr-in* of n is set to *null* in the next state of the system".

4 Complex Subsystems

Association *S-Link*
Participants s, t : *SubNet*
Connections
1. $(s.gateway.in = \mathsf{T}) \leftrightarrow (t.gateway.curr\text{-}out \neq null)$
2. $(s.gateway.out = \mathsf{T}) \leftrightarrow (t.gateway.curr\text{-}in = null)$
3. $(t.gateway.in = \mathsf{T}) \leftrightarrow (s.gateway.curr\text{-}out \neq null)$
4. $(t.gateway.out = \mathsf{T}) \leftrightarrow (s.gateway.curr\text{-}in = null)$
5. $\forall m \in \mathsf{message} : (s.gateway.send(m) \rightarrow t.gateway.get(m))$
6. $\forall m \in \mathsf{message} : (t.gateway.send(m) \rightarrow s.gateway.get(m))$

EndOfAssociation

Fig. 4. Association *S-Link*: An association between subsystems.

In the previous section, we allowed subsystems to be defined in terms of simpler subsystems. An important restriction is that the definition of subsystems cannot be cyclic. This is due to the fact that the semantics of a subsystem is based on the relativisation and inclusion of the presentations of the classes and subsystems of the lower layers, which of course is not possible if the subsystem dependency is cyclic.

In order to show how more complex subsystems can be defined, let us further extend the previous examples. Now that we have cells, links and subnets, we can define a network, composed of two subnets. In order to allow these subnets to communicate, we define a (higher level) association, as in Fig. 4. This association connects two subnets by connecting the corresponding gateways, in the same way *Link* connects cells. This is a generalisation of associations as defined in [1], since we now allow for subsystems to be participants, and not only basic components. Note also that multiple "dots" are employed to refer to attributes or actions of components within subsystems.

A network can now be easily defined as a higher level subsystem, as shown in Fig. 5. The first four axioms correspond to structural constraints. For instance, Axiom 3 indicates that subnets *sn1* and *sn2* are connected via a *S-Link* connection, while the network is active. A network can communicate with the outside world (e.g., other networks) through the *router* cell. Operations *conn* and *disconn* manage the access to the router for the subnets within the network (see Axioms 4-5). Finally, Axiom 6 indicates that two cells in different subnets cannot have the same address.

Subsystem *Network*
Attributes $\mathit{router}, \mathit{sn1}, \mathit{sn2}$: NAME
Actions $\mathit{conn}(\mathsf{NAME}),\ \mathit{disconn}(\mathsf{NAME})$
Axioms
1. $\Box[\mathit{Network} \rightarrow (\mathit{Cell}(\mathit{router}) \wedge \mathit{SubNet}(\mathit{sn1}) \wedge \mathit{SubNet}(\mathit{sn2}))]$
2. $\Box[\mathit{Network} \rightarrow \mathit{sn1} \neq \mathit{sn2}]$
3. $\Box[\mathit{Network} \rightarrow \mathit{S\text{-}Link}(\mathit{sn1}, \mathit{sn2})]$
4. $\Box[\forall s : \mathit{Network} \wedge \mathit{conn}(s) \rightarrow ((\neg \mathit{Link}(\mathit{router}, s.\mathit{gateway})) \wedge (\bigcirc \mathit{Link}(\mathit{router}, s.\mathit{gateway}))]$
5. $\Box[\forall s : \mathit{Network} \wedge \mathit{disconn}(s) \rightarrow$
$((\mathit{Link}(\mathit{router}, s.\mathit{gateway})) \wedge (\bigcirc \neg \mathit{Link}(\mathit{router}, s.\mathit{gateway}))]$
6. $\Box[\forall s_1, s2, n, m : \mathit{Network} \wedge \mathit{SubNet}(s_1) \wedge \mathit{SubNet}(s_2) \wedge$
$\mathit{Cell}(n, s_1) \wedge \mathit{Cell}(m, s_2) \wedge (s_1 \neq s_2) \rightarrow (n.\mathit{address} \neq m.\mathit{address})]$
EndOfSubsystem

Fig. 5. Subsystem *Network*: A higher level Subsystem Specification

It is clear from this example how the languages of the more basic subsystems and components are incorporated and used in a higher level subsystem. Again, the structurality property of the logic allows us to *promote* theorems from the lower layers of a specification to the upper layers. For instance, the subnet property

$$\Box[\forall n : \mathit{SubNet} \wedge \mathit{Cell}(n) \wedge \bigcirc(\neg \mathit{Cell}(n)) \rightarrow \mathit{rem\text{-}cell}(n)]$$

can be promoted into a network as:

$$\forall s : \Box[\forall n : \mathit{SubNet}(s) \wedge \mathit{Cell}(n, s) \wedge \bigcirc(\neg \mathit{Cell}(n, s)) \rightarrow \mathit{rem\text{-}cell}(n, s)].$$

We can then combine reasoning at different levels of a hierarchical specification to prove properties of the whole system. Unfortunately, due to space restrictions, we are unable to reproduce here a proof combining reasoning at all levels of a hierarchical specification.

5 Conclusion

We have presented an extension to a prototypical temporal specification language for specifying dynamic software architectures. The objective of the extension is to allow for hierarchical organisations of systems specifications, in terms of reconfigurable subsystems. We have generalised the concepts of association and subsystem, and shown an example illustrating the expressiveness of the extension.

One of the main characteristics of the formalism, which partially motivated our work, is the possibility of describing reconfigurable systems declaratively. Declarative specifications tend to be longer than operational ones; thus, mechanisms for modularising declarative specifications are crucial. The notion of subsystem complements the notion of basic component and allows us to further modularise specifications. Since components are hierarchically described, using classes and subsystems, as closed independent units, proofs can be localised to the relevant subparts of a specification.

We have already shown some evidence of the logic being expressive enough for specifying dynamic software architectures. We believe then that the logic is suitable as a "reasoning framework" for formal ADLs. Moreover, the logic could be used to provide a declarative semantics to some ADLs. Specifications in ADLs could be interpreted into the logic, and then the proof capabilities of it could be used in order to reason about properties of the specifications. We are currently exploring this line of work, mapping CommUnity [15] specifications into the logic in order to reason about properties of the specifications.

Even for simple systems, specifications tend to be large and complex. Although modularisation mechanisms help in alleviating the proof efforts, software tool support is a necessity. At the moment, we are experimenting with the use of the Stanford Temporal Prover (STeP) [4] in order to assist in the proofs in our logic.

References

[1] N. Aguirre and T. Maibaum, *A Logical Basis for the Specification of Reconfigurable Component-Based Systems*, in Proc. of FASE 2003, Warsaw, Poland, LNCS 2621, Springer, 2003.

[2] N. Aguirre and T. Maibaum, *Some Institutional Requirements for Temporal Reasoning about*

Dynamic Reconfiguration, to appear in Proc. of Symposium on Verification: Theory and Practice, Taormina, Italy, LNCS, Springer, 2003.

[3] R. Allen, R. Douence and D. Garlan, *Specifying and Analyzing Dynamic Software Architectures*, in Proc. of FASE 98, Lisbon, Portugal, LNCS, Springer, 1998.

[4] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma and T. Uribe, *Verifying Temporal Properties of Reactive Systems: a STeP Tutorial*, in Formal Methods in System Design, vol 16, 2000.

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

[6] J. Fiadeiro and A. Sernadas, *Structuring Theories on Consequence*, in D. Sannella and A. Tarlecki (eds), Recent Trends in Data Type Specification, LNCS 332, Springer, 1988.

[7] D. Garlan, *Software Architecture: A Roadmap*, in The Future of Software Engineering, A. Filkenstein (ed), ACM Press, 2000.

[8] D. Garlan, R. Monroe and D. Wile, *ACME: An Architecture Description Interchange Language*, in Proc. of CASCON'97, 1997.

[9] D. Garlan and D. Perry, *Software Architecture*. Panel Introduction. In Proc. of ICSE '94, Sorrento, Italy, 1994.

[10] J. Goguen and R. Burstall, *Institutions: Abstract Model Theory for Specification and Programming*, Journal of the ACM 39(1): 95-146, ACM Press, 1992.

[11] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, *Specifying Distributed Software Architectures*, in Proc. of ESEC'95, Sitges, Spain, LNCS 989, Springer, 1995.

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

[13] N. Medvidovic, *ADLs and Dynamic Architecture Changes*, in Proc. of ISAW 96, 1996.

[14] B. Meyer, *Object-Oriented Software Construction*, Second Edition, Prentice-Hall International, 2000.

[15] M. Wermelinger and J. Fiadeiro, *A Graph Transformation Approach to Software Architecture Reconfiguration*, in Science of Computer Programming 44, Elsevier, 2002.