A Logical Basis for the Specification of Reconfigurable Component-Based Systems*

Nazareno Aguirre** and Tom Maibaum

Department of Computer Science, King's College London
Strand, London, WC2R 2LS, United Kingdom
Tel: +44 207 848 1166, Fax: +44 207 848 2851
aguirre@dcs.kcl.ac.uk, tom@maibaum.org

Abstract. We present a logic and a prototypical specification language for specifying and reasoning about component-based systems with support for dynamic, i.e., run-time, architectural reconfiguration. We present the logic, an adaptation of an existing one proposed for specifying reactive systems, and some results that demonstrate its suitability for the specification of reconfigurable systems.
We then explicate how the specification language can be used to specify a reconfigurable (sub)system via layers defining component templates, association/connector templates and a layer specifying reconfiguration operations used to dynamically change the system architecture. We also illustrate the expressive power and proof capabilities of the logic.

1 Introduction

Due to the complexity and size of current software systems, the notion of structural architecture of systems, and its relationship to systems analysis and design, has come to play an important role in today's software development processes.

Special specification languages, called *architecture description languages* [10], were proposed to describe and analyse properties of (sometimes evolving) architectures. 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 [9]. While architecture description languages (ADLs) provide constructs for modelling the architecture of a system, they often do not support reasoning about possible system evolution. In other words, some ADLs support the definition of components, interconnections and transformation rules or operations for making architectures change dynamically, but any kind of reasoning about behaviours is often performed in some "meta-language", sometimes informally. Moreover, the description of architectural elements in ADLs, particularly those related to dynamic reconfiguration, is usually done in an operational way, as opposed to declaratively [4,6,12].

* 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

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 37–51, 2003.

Being able to specify and reason about the consequences of using certain reconfiguration operations in a declarative manner would add abstraction to what, to our understanding, can be operationally specified by ADLs. We therefore propose a temporal logic as a formal basis for the specification of reconfigurable systems. Temporal logic provides a declarative and well-known language to express behavioural properties, and is currently used in several branches of software engineering.

We adapt a logic proposed by Manna and Pnueli [7] for the specification of reactive systems for this purpose. We present the logic and some results that demonstrate its suitability for the specification of dynamically reconfigurable systems. A prototypical language based on this logic is defined, where systems specifications are hierarchically organised around the following notions:

- 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 be organised in a system;
- the notion of subsystem, the new notion that defines the (coarse grained) 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.

It is not our aim to propose another architecture description language, but to study an alternative declarative and formal semantics for software architectures. We prefer to illustrate the capabilites and expressive power of the formalism by defining a simple front-end to our logic (our prototypical language). This is simpler than trying to relate, at this stage of our work, our logic to existing high-level ADLs. In addition, it allows us to show how an interesting specification mechanism, hierarchical component organisation, can be achieved using our proposed formal basis.

2 The Logic

We start by describing the logic we use as a core for the specification of reconfigurable systems. This is a variant of a logic widely used for the specification of reactive systems, Manna-Pnueli logic [7,8]. A considerable amount of work has been done on the Manna-Pnueli logic, including the development of software tools for supporting the specification of systems [2].

Most of the definitions in this section are adapted from the ones in [7] and [11]. The use of the logic for expressing properties of systems is standard. The changes to the logic consist mainly in replacing the use of local variables by the use of, what we call, flexible function symbols, and by allowing some predicates to be interpreted in a state-dependent way. We need this in order to be able to specify reconfiguration.

2.1 Syntax

An *alphabet* (sometimes called *signature* or *vocabulary*) for this logic consists of: *(i)* a set $\mathbf{S}$ of sorts, *(ii)* a set of $(\mathbf{S}^* \times \mathbf{S})$-indexed flexible function symbols, *(iii)* a set of $(\mathbf{S}^* \times \mathbf{S})$-indexed rigid function symbols, *(iv)* a set of $(\mathbf{S}^*)$-indexed flexible predicate symbols, *(v)* a set of $(\mathbf{S}^*)$-indexed rigid predicate symbols and *(vi)* a countable set of $\mathbf{S}$-indexed variables.

Typed terms are constructed from the symbols of the vocabulary as usual. Formulae are constructed also in the usual way, using the traditional propositional connectives, equality, the unary temporal operators $\bigcirc$ and $\Diamond$, the binary temporal operator $\mathcal{U}$, and quantification over variables. For the definition of the semantics of the logic we consider $\neg$ and $\rightarrow$ as the only propositional connectives, since the others can be obtained from these.

Terms are used to denote individuals, i.e., elements of the universe of discourse, and operations on them. For the specification language (and the application domain) we are interested in, we will need, for instance, terms to denote integers, strings, booleans, etc, and the usual operations on them.

The intended meaning of propositional connectives is the standard. Having in mind that validity of formulae (in a model) will be subject to a (current) state, and that states are linearly organised, the intended meanings of $\bigcirc\alpha$ and $\alpha\mathcal{U}\beta$ are "α is true in the next state" and "α is true (at least) until β becomes true", respectively. This is formalised in the next section.

2.2 Semantics

First, let us introduce some definitions, necessary in order to give the semantics of this logic.

Definition 1. *Given an alphabet $\mathcal{A}$, a (semantic) structure M for it is a mapping that assigns:*

- *for each sort S in $\mathcal{A}$, a set S_M,*
- *for each rigid function symbol $f : S_1, \ldots, S_k \rightarrow S$, a function*

$$f_M : S_{1_M}, \ldots, S_{k_M} \rightarrow S_M,$$

- *for each rigid predicate symbol $p : S_1, \ldots, S_k$, a relation*

$$p_M \subseteq S_{1_M} \times \ldots \times S_{k_M}.$$

Given an alphabet $\mathcal{A}$ and an $\mathcal{A}$-structure M, a *state* is a function s that maps:

- every flexible function symbol $f : S_1, \ldots, S_k \rightarrow S$ to a function

$$f_M : S_{1_M}, \ldots, S_{k_M} \rightarrow S_M,$$

- every flexible predicate symbol $p : S_1, \ldots, S_k$ to a relation

$$p_M \subseteq S_{1_M} \times \ldots \times S_{k_M}.$$

A *trajectory* σ in an $\mathcal{A}$-structure M is an infinite list of states. Given a trajectory $\sigma = s_0, s_1, \ldots$, we denote by $\sigma^{(k)}$ the suffix $s_k, s_{k+1}, \ldots$.

An *assignment* for an $\mathcal{A}$-structure is a mapping that, for every variable $V : S$, assigns a value $a \in S_M$. Given an assignment A, a variable $x : S$ and a value $d \in S_M$, we denote by $A_{x \lhd d}$ the assignment that coincides with A for all variables $y \neq x$, and that maps x to d.

Definition 2. *Let $\mathcal{A}$ be an alphabet. An* interpretation *is a triple $I = (M, A, \sigma)$, where M is a $\mathcal{A}$-structure, A is an assignment for M, and σ is a trajectory in M.*

Given an interpretation $I = (M, A, \sigma)$, we denote by $I_{x \lhd d}$ the interpretation $(M, A_{x \lhd d}, \sigma)$, and by $I^{(k)}$, for a natural number k, the interpretation $(M, A, \sigma^{(k)})$.

Given an interpretation $I = (M, A, \sigma)$ for $\mathcal{A}$, we define $I(t)$, for a $\mathcal{A}$-term t, as follows:

- for a constant c, $I(c) = c_M$,
- for a variable v, $I(v) = A(v)$,
- for a rigid 0-ary function symbol $f : S$, $I(f) = f_M$,
- for a flexible 0-ary function symbol $f : S$, $I(f) = \sigma_0(f)$, where σ_0 is the first state in the trajectory σ,
- for a term $f(t_1, \ldots, t_k)$, where f is a rigid function symbol,

$$I(f(t_1, \ldots, t_k)) = f_M(I(t_1), \ldots, I(t_k)),$$

- for a term $f(t_1, \ldots, t_k)$, where f is a flexible function symbol,

$$I(f(t_1, \ldots, t_k)) = \sigma_0(f)(I(t_1), \ldots, I(t_k)),$$

We are ready to define *satisfaction* of a formula under a given interpretation.

Definition 3. *Let $\mathcal{A}$ be an alphabet, and $I = (M, A, \sigma)$ an interpretation. We define satisfaction of a formula α (over alphabet $\mathcal{A}$) in I, in symbols $\overset{I}{\models} \alpha$, as follows:*

- $\overset{I}{\models} t_1 = t_2$ *if and only if* $I(t_1) = I(t_2)$,
- $\overset{I}{\models} p(t_1, \ldots, t_k)$, *for a rigid predicate p, if and only if* $(I(t_1), \ldots, I(t_k)) \in p_M$
- $\overset{I}{\models} p(t_1, \ldots, t_k)$, *for a flexible predicate p, if and only if*

$$(I(t_1), \ldots, I(t_k)) \in \sigma_0(p)$$

- $\overset{I}{\models} \neg\beta$ *if and only if it is not the case that* $\overset{I}{\models} \beta$,
- $\overset{I}{\models} \beta_1 \rightarrow \beta_2$ *if and only if* $\overset{I}{\models} \neg\beta_1$ *or* $\overset{I}{\models} \beta_2$,
- $\overset{I}{\models} \bigcirc\beta$ *if and only if* $\overset{I^{(1)}}{\models} \beta$,

- $\overset{I}{\models} \Diamond\beta$ *if and only if there exists a* $k \geq 0$ *such that* $\overset{I^{(k)}}{\models} \beta$,
- $\overset{I}{\models} \beta_1 \mathcal{U} \beta_2$ *if and only if for some* $k \geq 0$, $\overset{I^{(k)}}{\models} \beta_2$ *and for all* $0 \leq i < k$, $k \geq 0$, $\overset{I^{(i)}}{\models} \beta_1$
- $\overset{I}{\models} \forall x \in S : \beta$ *if and only if for all* $d \in S_M$ *it is the case that* $\overset{I_{x \lhd d}}{\models} \beta$

Given a set of formulae Φ over an alphabet $\mathcal{A}$, an $\mathcal{A}$-interpretation I is called a *model* of Φ if and only if $\overset{I}{\models} \phi$, for all ϕ in Φ.

Semantic Consequence. We overload the symbol $\models$, using it now for defining a relation between sets of formulae over a signature $\mathcal{A}$. Let $\mathcal{A}$ be a signature, Φ and Ψ sets of formulae over $\mathcal{A}$. Then, we say that Ψ is a *semantic consequence* of Φ wrt $\mathcal{A}$, in symbols $\Phi \models_{\mathcal{A}} \Psi$, if and only if for every interpretation I wrt $\mathcal{A}$, if I is a model of Φ then it is also a model of Ψ. We drop the subscript of $\models$ when it is clear from the context.

Proposition 1. *The binary relation* $\models$ *of semantic consequence between sets of formulae over a signature* $\mathcal{A}$ *has the following properties:*

- *Reflexivity:* $\Phi \models \Phi$
- *Cut: if* $\Phi \cup \Phi_1 \models \Psi$ *and* $\Phi \models \phi$, *for all* ϕ *in* Φ_1, *then* $\Phi \models \Psi$
- *Monotonicity: if* $\Phi \models \Psi$, *then* $\Phi \cup \Phi_1 \models \Psi$,

for all sets Φ, Φ_1, Ψ *of formulae over* $\mathcal{A}$.

2.3 A Proof Calculus

A sound proof calculus for the logic can be obtained by easily adapting the axioms for the Manna-Pnueli logic presented in [11]. Inference rules presented there also preserve validity in our adaptation; however, the definition of valid substitutability has to be changed, to reflect the fact that, in our adaptation, flexible predicate and function symbols are state-dependent, which has an impact in the quantification axioms. For the reader aware of [11], these changes consist only of adapting the definition of predicate $globsub(t, x, w)$ in [11] pp. 165–161, whose intended meaning is "the replacement of x by t in formula w does not generate new occurrences of flexible symbols within the scope of temporal operators and no new ocurrences of bound variables".

The resulting proof-theoretical consequence relation satisfies reflexivity, cut and monotonicity.

2.4 Signature and Theory Morphisms

It will be necessary for us to combine different alphabets and formulae in order to be able to build specifications. For this purpose, we need the logic to satisfy certain structural properties.

Definition 4. *A* signature morphism σ *between signatures* $\mathcal{A}$ *and* $\mathcal{B}$ *as a function that maps:*

- *each sort in* $\mathbf{S}_{\mathcal{A}}$ *to a sort in* $\mathbf{S}_{\mathcal{B}}$,
- *each flexible (resp. rigid) function symbol* $f : S_1, \ldots, S_k \to S$ *in* $\mathcal{A}$ *to a flexible (resp. rigid) function symbol* $\sigma(f) : \sigma(S_1), \ldots, \sigma(S_k), S'_{k+1}, \ldots, S'_n \to \sigma(S)$ *in* $\mathcal{B}$,
- *each flexible (resp. rigid) predicate symbol* $p : S_1, \ldots, S_k$ *in* $\mathcal{A}$ *to flexible (resp. rigid) predicate symbol* $\sigma(p) : \sigma(S_1), \ldots, \sigma(S_k), S'_{k+1}, \ldots, S'_n$ *in* $\mathcal{B}$,
- *each variable* $x : S$ *in* $\mathcal{A}$ *to a variable* $\sigma(x) : \sigma(S)$ *in* $\mathcal{B}$.

Note that function and predicate symbols could be mapped to symbols with a greater arity. This is crucial for the way we deal with reconfiguration.

Having defined mappings of symbols from an alphabet to another, we define how to translate formulae from one alphabet to another, in a way that is useful for the purpose of specifying reconfigurable systems.

Definition 5. *Let* $\sigma : \mathcal{A} \to \mathcal{B}$ *be a signature morphism. The function* $Gr_{\sigma} : L_{\mathcal{A}} \to L_{\mathcal{B}}$, *is defined as follows: Given a formula* α, *the formula* $Gr_{\sigma}(\alpha)$ *is the result of translating the symbols in* α *using* σ, *placing fresh variables in the free spaces resulting from translating function or predicate symbols into others of a greater arity, and quantifying these universally.*

Example 1. Consider the formula $\Box[(\exists x \in S : p(x)) \to q]$. If a signature morphism maps S to S', $p : S$ to $p' : S', S''$, q to $q' : S''$, and $x : S$ to $x : S'$. Then, the formula resulting from the translation is:

$$\forall y \in S'' : [\Box[(\exists x \in S' : p(x, y)) \to q(y)]]$$

Theorem 1. *Let* $\sigma : \mathcal{A} \to \mathcal{B}$ *be a signature morphism, and* Φ *and* Ψ *be sets of* $\mathcal{A}$*-formulae. Then,*

- $\Phi \models_{\mathcal{A}} \Psi$ *implies* $Gr_{\sigma}(\Phi) \models_{\mathcal{B}} Gr_{\sigma}(\Psi)$.
- $\Phi \vdash_{\mathcal{A}} \Psi$ *implies* $Gr_{\sigma}(\Phi) \vdash_{\mathcal{B}} Gr_{\sigma}(\Psi)$.

These results imply that this logic constitutes a π-institution [5], both considering semantic and proof-theoretic consequence.

3 The Language

Here we present a prototypical language, in which we make use of the logic defined in the previous section for specifying some standard elements found in ADLs. The language is inspired by the language defined in [3], and it is greatly influenced by ideas from CommUnity [12]. In particular, the definition of associations is based on the idea of coordination.

As we already indicated, it is not our aim to introduce yet another ADL. We use the language defined here just to illustrate the expressive power of the

logic, and the mechanisms applied to represent dynamic reconfiguration. It also allows us to show how our declarative setting makes possible to provide some interesting features.

We start by considering the specification of datatypes. A datatype specification is simply a theory presentation (i.e., a finite set of formulae) over a signature $\mathcal{ADT}$ with no flexible predicate or function symbols. We assume this specification contains the definition of all standard datatypes, such as integers, sequences, natural numbers, etc. In addition, we assume that a type NAME is defined, with a sufficiently large set of constants, and no operations. Let us denote this theory presentation by:

$$\mathcal{ADT} = \langle (S_{\mathcal{ADT}}, \emptyset, Fun^r_{\mathcal{ADT}}, \emptyset, Pred^r_{\mathcal{ADT}}, Var_{\mathcal{ADT}}), Ax_{\mathcal{ADT}} \rangle$$

3.1 Class Definitions

The basic building blocks of specifications in our prototypical language are components. Component templates are specified by means of *class definitions*. A class definition consists simply of: *(i)* finite sets of attributes and read variables whose type is a sort defined in $\mathcal{ADT}$ excluding NAME, *(ii)* a finite set of actions, which can have arguments typed with sorts in $\mathcal{ADT}$ excluding NAME. A class specification is equipped with a set of formulae over the signature:

$$(S_{\mathcal{ADT}} - \{\mathsf{NAME}\}, Rv \cup Att, Fun^r_{\mathcal{ADT}}, Act, Pred^r_{\mathcal{ADT}}, Var_{\mathcal{ADT}}),$$

where Rv, Att and Act denote the sets of attributes, read variables and actions respectively. That is to say, we use read variables and attributes as flexible function symbols, and actions as flexible predicates, extending the vocabulary defined in the datatype specification. Axioms in the class specification are not allowed to use datatype NAME, since it will serve a special purpose later on. The purpose of the axioms of a class specification is to describe the meaning of the actions, i.e. their effect on attributes.

Example 2. Consider the class specification in Fig. 1. It is the specification of a producer; intuitively, the first formula indicates that when action *p-init*() (meant to "initialise" the component) is executed, the attribute *p-waiting* is set to F (false). The second formula says that in order to be able to perform the *produce* operation, the component must be not waiting. The third formula expresses that *produce*(x) causes the component to be waiting and the attribute *p-current* to be set to x in the next state. Formulae 4, 5 and 6 indicate how action *send*() works, calling action *dispatch*. Finally, formula 7 says that action *dispatch* can only be called by *send*, i.e., it cannot occur spontaneously.

It is worth noting that type item is a basic type; we do not use any particular features of elements of this type, we just assume is not a class type, i.e. item is defined in $\mathcal{ADT}$.

Class *Producer*
Read Variables: *ready-in* : boolean
Attributes: *p-current* : item, *p-waiting* : boolean
Actions: *produce*(x : item), *send*(), *dispatch*(x : item), *p-init*()
Axioms

1. $\Box[p\text{-}init() \rightarrow (p\text{-}waiting = \mathsf{F})]$
2. $\Box[\forall x \in \mathsf{item} : produce(x) \rightarrow (p\text{-}waiting = \mathsf{F})]$
3. $\Box[\forall x \in \mathsf{item} : produce(x) \rightarrow \bigcirc((p\text{-}waiting = \mathsf{T}) \wedge (p\text{-}current = x))]$
4. $\Box[send() \rightarrow ((ready\text{-}in = \mathsf{T}) \wedge (p\text{-}waiting = \mathsf{T}))]$
5. $\Box[send() \rightarrow dispatch(p\text{-}current)]$
6. $\Box[send() \rightarrow \bigcirc(p\text{-}waiting = \mathsf{F})]$
7. $\Box[[\exists x \in \mathsf{item} : dispatch(x)] \rightarrow send()]$

EndofClass

Fig. 1. Class specification *Producer*

Semantics of Class Definitions. A class specification is interpreted as a theory presentation, over the signature

$$(S_{\mathcal{ADT}}, Rv \cup Att, RF_{\mathcal{ADT}}, P_{\mathcal{ADT}} \cup Act, Var_{\mathcal{ADT}}).$$

The axioms of the presentation are obtained by putting together: *(i)* the axioms explicitly provided for the class definition, *(ii)* the axioms of the datatype specification, *(iii)* a special (implicit) axiom, called the *locality axiom* for the specification, whose general form is:

$$\Box\left[\left(\bigvee_{g \in Act} \exists \overline{x}_g : g(\overline{x}_g)\right) \vee \left(\bigwedge_{a \in Att} \bigcirc(a) = a\right)\right]$$

where Act and Att are the sets of actions and attributes of the component, respectively. The intuitive meaning of the locality axiom, originally proposed in [3], is: "in every state it is the case that either one of the actions is executed, or all the attributes remain unchanged".

Note that read variables are not considered in the locality axiom; this is because read variables are special attributes, meant to be "entry points" used by a component to query the state of the environment; therefore, they are not controlled by the component, which implies they could change, from the point of view of the component, arbitrarily.

The inclusion of the axioms defining datatypes in the theory of a component definition justifies the following Theorem:

Theorem 2. *Given a class definition C, its corresponding theory is an extension of the theory of the datatype specification $\mathcal{ADT}$.*

3.2 Associations

Once classes have been defined, ways of making components interact can be defined. This is done by means of what we call *associations*. Associations are simply templates of *connectors*, in the sense of [1]. The syntax is very simple: An association consists of:

- a name for the association (distinct from names of other linguistic elements already defined),
- a set of participants, typed with class names,
- a set of connections, which are expressions of the form:

$$x.\alpha \leftrightsquigarrow y.\beta$$

where x and y are participants of classes A and B respectively, and α and β are either:

- formulae in the languages of A and B respectively, or
- terms of the same sort, in the languages of A and B respectively.

The intended meaning of an association definition is that, whenever certain instances are related using an association instance, then they are forced to synchronise as the connections indicate.

The actual interpretation of associations, as special formulae, takes place at the next level of the specification, the subsystems.

Class *Consumer*
Read Variables: *ready-ext* : boolean
Attributes: *c-current* : item, *c-waiting* : boolean
Actions: *consume*(*x* : item), *extract*(*x* : item), *c-init*()
Axioms

1. $\Box[c\text{-}init() \rightarrow (c\text{-}waiting = \mathsf{T})]$
2. $\Box[\forall x \in \mathsf{item} : extract(x) \rightarrow ((c\text{-}waiting = \mathsf{T}) \wedge (ready\text{-}ext = \mathsf{T}))]$
3. $\Box[\forall x \in \mathsf{item} : extract(x) \rightarrow \bigcirc((c\text{-}waiting = \mathsf{F}) \wedge (c\text{-}current = x))]$
4. $\Box[\forall x \in \mathsf{item} : consume(x) \rightarrow \bigcirc(c\text{-}waiting = \mathsf{T})]$
5. $\Box[\forall x \in \mathsf{item} : consume(x) \rightarrow ((c\text{-}current = x) \wedge (c\text{-}waiting = \mathsf{F}))]$
6. $\Box[c\text{-}waiting = \mathsf{F} \rightarrow \Diamond(consume(c\text{-}current))]$

EndofClass

Fig. 2. Class specification *Consumer*

Example 3. Consider the class specifications in Figs. 1 and 2; we define the association in Fig. 3 with the intention of making producers and consumers interact. The intuitive meaning of the connections is that, whenever a producer p is connected to a consumer c by means of a connector *Prods-for*, then:

- the read variable *ready-in* in p is identified with the attribute *c-waiting* of c (and viceversa),
- the attribute *p-waiting* of p is identified with the read variable *ready-ext* of c (and viceversa),
- whenever $dispatch(x)$ occurs in p, $extract(x)$ also occurs (and viceversa).

Note the last axiom of the *Consumer* specification. It expresses a *liveness* condition on consumers. This shows the expressiveness of the logic, but it also shows how "low-level" the language is in its present status, since this kind of properties can be directly enforced as axioms[1].

Association *Prods-for*
Participants: $p : Producer$, $c : Consumer$
Connections:

$p.ready\text{-}in \leftrightsquigarrow c.c\text{-}waiting$
$p.p\text{-}waiting \leftrightsquigarrow c.ready\text{-}ext$
$p.dispatch(x) \leftrightsquigarrow c.extract(x)$

EndofAssociation

Fig. 3. Association specification *Prods-for*

3.3 Subsystems

This is the upper layer of the language. Once class and association specifications are given, a subsystem can be declared. Intuitively, we can describe a subsystem as a complex component, built out of instances of classes (basic components) inter-related by means of connectors, i.e., instances of associations. The key capability of a subsystem, what motivated the definition of the logic, is that it can have operations that dynamically change its architectural state.

A subsystem *Sub* then has a finite set of actions, whose arguments must be of types defined in $\mathcal{ADT}$, now including NAME. Also, as for class definitions, we use logical axioms for specifying the behaviour of a subsystem. The alphabet $\mathcal{A}_{Sub}$ over which the subsystem formulae are expressed is composed of:

[1] Liveness properties are an example of properties that might not be preserved when a component becomes part of certain systems. In our language, these properties can be expressed, and because of the semantics for subsystems, they will be "preserved" in any subsystem. When, because of some reason, such a property is not preserved, this will be reflected as an inconsistency in the theory of the corresponding subsystem. Although we agree this is certainly not the best way of "catching" the anomaly, we believe this matter can be solved by defining a more suitable "front-end" to the logic. However, we choose at the moment to illustrate the expressive power of the logic, and not to be concerned about the specification language definition, something orthogonal to the main ideas of our work.

- the sorts defined in $\mathcal{ADT}$,
- the rigid function and predicate symbols defined in $\mathcal{ADT}$,
- the flexible function and predicate symbols resulting from class definitions, adding to all of them an extra parameter of sort NAME,
- a flexible predicate symbol $A : \mathsf{NAME}$ for each class definition A,
- a flexible predicate symbol $R : \underbrace{\mathsf{NAME}, \ldots, \mathsf{NAME}}_{k \text{ times}}$ for each association definition R with k participants
- a flexible predicate symbol $a : S_1, \ldots, S_k$ for each subsystem action of type $a(x_1 : S_1, \ldots, x_k : S_k)$.

Constants of type NAME represent names of instances. Predicate $A : \mathsf{NAME}$, for a class definition A, is meant to characterise the names of live instances of type A in each state. Predicate R for an association R is used to denote the instances of connectors in each state. Clearly, there exists a signature morphism $\sigma_{A \leq Sub}$ from the signature of every class definition A into the signature described above.

Example 4. Consider the subsystem specification in Fig. 4. It is a description of a complex component, built out of instances of producers and consumers interrelated by connectors of the kind defined by *Prods-for*. The subsystem consists of four operations: *(i) init*(), which is specified by axioms 1-6 (it is meant to be an initialisation operation), *(ii) change*(y), specified by axioms 7-10, makes the only producer of the subsystem, P, to produce for consumer y, *(iii) create*(y), which creates a new consumer, and *(iv) del*(y), which deletes an existing consumer. When the sort of a variable is not explicitly indicated in a quantification of a formula, we consider it to be NAME.

Operation *init*() creates a producer, P, and a consumer, C. It can be called only once, as specified by axiom 5. Operation *change* is the only one that can change the interconnections, besides the initialisation.

Note that the extra parameter of flexible symbols from component definitions is denoted using the "dot" notation from object orientation, making it more readable. As can be seen, this causes operations and attributes declared in classes to be "relativised" to a corresponding instance name. It is clear from the example how the language of the components is incorporated into the language of a subsystem. The extra parameter added to flexible symbols indicates to which instance the action or attribute corresponds to. For example, if we see axiom 4, it indicates that *init*() in the subsystem "calls" the initialisation operations of P and C, now denoted by $P.p\text{-}init()$ and $C.c\text{-}init()$, respectively.

Semantics of Subsystems. As for basic components, we interpret subsystem specifications as theory presentations. A subsystem *Sub* describes a theory presentation over signature $\mathcal{A}_{Sub}$, whose axioms are:

- The formulae explicitly provided in the subsystem specification,

Subsystem Multiple_Consumers
Operations: $init()$, $change(x : \mathsf{NAME})$, $create(x : \mathsf{NAME})$, $del(x : \mathsf{NAME})$
Axioms

1. $\Box[init() \rightarrow \bigcirc(Producer(P) \wedge Consumer(C) \wedge Prods\text{-}for(P, C))]$
2. $\Box[init() \rightarrow \bigcirc(\forall x : Producer(x) \rightarrow x = P)]$
3. $\Box[init() \rightarrow \bigcirc(\forall y : Consumer(y) \rightarrow y = C)]$
4. $\Box[init() \rightarrow \bigcirc(P.p\text{-}init() \wedge C.c\text{-}init())]$
5. $\Box[init() \rightarrow \bigcirc(\Box(\neg init()))]$
6. $\Box[\neg init() \rightarrow \forall x : (Producer(x) \leftrightarrow \bigcirc(Producer(x)))]$
7. $\Box[\forall y : change(y) \rightarrow (\neg Prods\text{-}for(P, y)) \wedge \bigcirc(Prods\text{-}for(P, y))]$
8. $\Box[\forall y : change(y) \rightarrow [\exists y' : Prods\text{-}for(P, y') \rightarrow \neg \bigcirc (Prods\text{-}for(P, y'))]]$
9. $\Box[\forall y : change(y) \rightarrow P.p\text{-}waiting = \mathsf{F}]$
10. $\Box[\forall y : \neg change(y) \rightarrow (Prods\text{-}for(P, y) \leftrightarrow \bigcirc(Prods\text{-}for(P, y)))]$
11. $\Box[\forall y : create(y) \rightarrow (\neg Consumer(y)) \wedge \bigcirc(Consumer(y))]$
12. $\Box[\forall y : [(\neg Consumer(y)) \wedge \bigcirc(Consumer(y))] \rightarrow create(y) \vee init()]$
13. $\Box[\forall y : create(y) \rightarrow \bigcirc(y.c\text{-}init())]$
14. $\Box[\forall y : del(y) \rightarrow (Consumer(y)) \wedge \bigcirc(\neg Consumer(y))]$
15. $\Box[\forall y : [(Consumer(y)) \wedge \bigcirc(\neg Consumer(y))] \rightarrow del(y)]$

EndofSubsystem

Fig. 4. A subsystem specification

- The formulae corresponding to every class definition A, appropriately translated into the language of $\mathcal{A}_{Sub}$ by $Gr_{\Box_{A \triangleleft Sub}}$ (see Example 1, showing how a formula similar to Axiom 7 in *Producer* is translated to be incorporated in a subsystem),
- Implicit formulae characterising association definitions,
- Implicit formulae characterising general properties of subsystems.

We have chosen to have mutually exclusive sets of symbols for the component definitions (except, of course, for the symbols corresponding to the datatypes specification) to simplify the presentation.

Characterisation of Associations. We characterise association definitions by means of formulae that are incorporated in the theory of a subsystem. These formulae indicate the "type" of the arguments of the connectors, and how they communicate, according to the connections defined in the association. For our example, the formulae are:

$$\Box[\forall x, y : Prods\text{-}for(x, y) \rightarrow Producer(x) \wedge Consumer(y)]$$
$$\Box[\forall x, y : Prods\text{-}for(x, y) \rightarrow x.ready\text{-}in = y.c\text{-}waiting]$$
$$\Box[\forall x, y : Prods\text{-}for(x, y) \rightarrow x.p\text{-}waiting = y.ready\text{-}ext]$$
$$\Box[\forall x, y : Prods\text{-}for(x, y) \rightarrow \forall i : \mathsf{item} : x.dispatch(i) \leftrightarrow y.extract(i)]$$

General Properties of Subsystems. Besides the formulae characterising associations, there are other general properties that are specified by means of implicit

formulae in a subsystem. These formulae indicate, for example, that nothing can be at the same time an instance of two different classes, that operations of "dead" instances cannot take place, that a subsystem may evolve only by means of its own operations (locality of a subsystem), etc. We list here a few to show how they are expressed:

$$\Box[\forall x : \forall i \in \mathsf{item} : x.dispatch(i) \rightarrow Producer(x)]$$
$$\Box[\forall x : \neg(Producer(x) \wedge Consumer(x))]$$

The justification for the need of flexible predicates is their use to denote action occurrence, which should clearly be state-dependent. It is not sufficient to have flexible propositional variables, since we want to be able to deal with parameterised actions (for instance, note that when building subsystems, actions from basic components need to be relativised to instance names, which requires the consideration of a new parameter of type NAME for the corresponding predicate). For the case of function symbols, it is not sufficient to have local variables as in [7] because attributes of classes (program variables) have to be "relativised" to instance names when considered in a subsystem; therefore, flexible 0-ary functions (local variables) generated by a class A have to be represented by flexible unary functions in an including subsystem *Sub*.

The way the theory of a subsystem is constructed, importing axioms from the class specifications, and the results in Theorem 1 justify the following:

Theorem 3. *The theory corresponding to a subsystem definition Sub is an extension of the translation* $Gr_{\blacksquare_{A \cdot Sub}}(\Phi_A)$ *of the theory* Φ_A *of every class definition A.*

Some Properties of the Subsystem. Here we provide some properties of the subsystem of Fig. 4, illustrating the expressive power of the logic. We also include a sketch of the proof of one of these properties. It is worth mentioning that these properties can be proved using our adapted version of the proof calculus for the Manna-Pnueli logic, although we cannot include the complete detailed proofs due to space restrictions.

Property 1: "After the subsystem has been initialised, P is always producing for some consumer". We can express this as follows:

$$\Box[init() \rightarrow \bigcirc(\Box(\exists y : Prods\text{-}for(P, y)))]$$

Proof: We can prove this property by using one of the proof methods for invariance described in [8]. First we prove that $init() \rightarrow \bigcirc(\exists y : Prods\text{-}for(P, y))$, which follows from Axiom 1 in the Subsystem. Then we prove that

$$init() \rightarrow \bigcirc(\exists y : Prods\text{-}for(P, y) \rightarrow \bigcirc(\exists y' : Prods\text{-}for(P, y')))$$

i.e., that after $init()$, the property is preserved by all the actions. This follows from the fact that action *change* is the only one that can "disconnect" consumers (Axiom 10 in the subsystem), and when it does, it connects P to another consumer (Axiom 7 in the subsystem), reestablishing the property.

Property 2: "After the subsystem has been initialised, if P has produced an item, the connected consumer(s) will remain to be connected to it until the item is dispatched". We express this in the following way:

$$\Box[\mathit{init}() \rightarrow \bigcirc(\Box(\exists i \in \mathsf{item} : P.\mathit{produce}(i) \rightarrow \\ \forall y : \mathit{Prods\text{-}for}(P, y) \rightarrow \mathit{Prods\text{-}for}(P, y)\, \mathcal{U}\, P.\mathit{dispatch}(i)))]$$

This property involves the combination of axioms explicitly provided in the subsystem (such as Axiom 9) with properties of the components (such as Axiom 6 in the producer specification).

Property 3: "After the subsystem has been initialised, P is the only instance of *Producer*". This is expressed as follows:

$$\Box[\mathit{init}() \rightarrow \bigcirc(\Box(\forall x : \mathit{Producer}(x) \leftrightarrow x = P))].$$

4 Related Work

Our motivation is to formally reason about behavioural properties of dynamically reconfigurable systems. Our work is then related to approaches to the specification of software architectures, such as the work on various ADLs [1,4,9, 10].

This work is specially related to approaches to formal specificatiton of dynamic reconfiguration, such as those based on graph grammars [12] and chemical abstract machines [6]. As we indicated before, these approaches propose operational languages for specifying reconfigurable systems. Our logic provides a declarative, and therefore more abstract, framework for characterising reconfigurable systems. Moreover, reasoning can be performed within the formalism, as opposed to doing so in some meta-language, as in the mentioned alternatives.

There are many similarities between the work presented here and the work on CommUnity [12], since both are based on the logical and semantic foundations of [3]. In particular, the idea of coordination as a mechanism for achieving communication is used in our prototypical specification language for specifying associations.

The logic we presented is an adaptation of the Manna-Pnueli logic [7,8]. Unfortunately, we are not able to use the original logic for the purpose of specifying reconfigurable systems, due to a restriction on the type of flexible symbols allowed, i.e., only flexible constants/variables are allowed. Our adaptation of the logic overcomes this difficulty, but means that the extensive work on the Manna-Pnueli logic not fully applicable to our formalism. However, most of the work is relevant to our proposed adaptation. For instance, the original logic coincides with our adaptation in the fragment used for specifying datatypes and basic components; therefore, the tool support for the Manna-Pnueli logic can be used as is for assisting the verification of these parts of a specification. We believe it is not difficult to adapt the existing tool support to cope with subsystems, which require flexible function and predicate symbols. The extension also requires STeP [2] to be able to deal with structured theories and the exporting of theorems from a component to a subsystem.

5 Conclusions

We presented a logic suitable for specifying and reasoning about component-based systems with support for run-time architectural reconfiguration. The suitability of the logic is illustrated by the definition of a prototypical specification language on top of it, which shows the expressive power of the logic. The way in which associations are represented in the language 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.

We think our work complements the ones regarding ADLs such as [12,1,6], by providing a uniform language to state and prove properties, that could then be related to specifications in a number of different ADLs.

Among the future work related to the logic presented here, we are studying ways of defining suitable inheritance relationships between components, which might allow us to have polymorphic reconfiguration operations in subsystems. We also want to study how specifications in some ADLs can be interpreted in our logic, to provide them with a formal logical semantics and proof capabilities.

References

1. R. Allen and D. Garlan, *Formalizing Architectural Connection*, in Proceedings of the 16th International Conference on Software Engineering ICSE '94, Sorrento, Italy, 1994.
2. 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.
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. D. Garlan, R. Monroe and D. Wile, *ACME: An Architecture Description Interchange Language*, in Proc. of CASCON'97, 1997.
5. J. Goguen and R. Burstall, *Institutions: Abstract Model Theory for Specification and Programming*, in Journal of the ACM (JACM), vol. 39, No. 1, 1992.
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. Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems – Safety*, Springer, 1995.
9. N. Medvidovic, *ADLs and Dynamic Architecture Changes*, in Proceedings of the Second Int. Software Architecture Workshop (ISAW-2), 1996.
10. N. Medvidovic and R. Taylor, *A Framework for Classifying and Comparing Architecture Description Languages*, In ESEC-FSE'97, 1997.
11. J. Ostroff, *Temporal Logic for Real-Time Systems*, Advanced Software Development Series, Research Studies Press Ltd, John Wiley and Sons, 1989.
12. M. Wermelinger, A. Lopes and J. Fiadeiro, *A Graph Based Architectural (Re)configuration Language*, in ESEC/FSE'01, V.Gruhn (ed), ACM Press, 2001.