# Some Institutional Requirements for Temporal Reasoning on Dynamic Reconfiguration of 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 study a logic adapted for the purpose of specifying component based systems with support for run time reconfiguration. In particular, we analyse some institutional properties of this logic, related to compositional reasoning in specifications.
The logic is an adaptation of the Manna-Pnueli logic, a first-order temporal logic originally proposed to describe reactive systems. We present our variant in detail, and motivate the required extensions by showing how reconfigurable systems can be specified and how reasoning can be carried out in the presence of these properties. Some issues regarding the use of STeP for proof support are discussed.

## 1 Introduction

When the complexity of software systems started to increase some decades ago, in part due to more complex or bigger application domains, the need for techniques that would allow developers to modularise or divide systems and the problems they solve into manageable parts became crucial. Various heuristic techniques regarding modularisation were conceived. Some of these then evolved to become constructs of what were at that time modern programming languages, and eventually were integrated into programming methodologies [22][23]. The advantages that structuring software systems into *modules* has in all phases of software development, from analysis to maintenance, were instantly recognised and have strengthened over the intervening decades.

In the past decade, a new branch of software engineering emerged with the name software architectures [4][11]. This branch (re)emphasises the notion of

[*] 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.

module, or component, at a perhaps higher level of abstraction than that used in other modern modelling (and programming) methodologies, such as object orientation [20]. Software architectures suggest the modelling of systems structure in terms of components related by means of *connectors*, thus introducing a second modularisation concept to accompany that of components. The motivating principles are the very same ones originally motivating modularisation techniques. The increasing attention in higher level structural descriptions of systems led to the development of a special type of specification languages, called *architecture description languages* (ADLs) [19]. The purpose of ADLs is to describe and analyse properties of software architectures.

Modern applications typically require a feature that some ADLs are able to deal with, namely *dynamic reconfiguration*, i.e., the run time modification of the system's structure [18]. Although this is not an inherent feature of software architectures, it appears frequently and naturally, perhaps due to the success of object oriented methodologies and programming languages, where it is intrinsic. While ADLs provide constructs for modelling the architecture of a system, they often do not support within the language reasoning about possible system evolution. More precisely, 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", often 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 [12][14][24].

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 proposed a temporal logic based formalism for the specification of reconfigurable systems [2][3]. Temporal logic provides a declarative and well known language to express behavioural properties, and is currently used in several branches of software engineering. Moreover, the use of temporal logic as a formal basis for software architectures leads to direct support for reasoning. In fact, there exists currently tool support for the temporal logic we based our work on, the Manna-Pnueli logic [16]. Although this tool, the Stanford Temporal Prover (STeP) [6], does not fully meet our needs, it can still be used to assist in the reasoning regarding reconfigurable systems.

We adapted the Manna-Pnueli logic [16], originally proposed for specifying reactive systems, for the purpose of describing reconfigurable component based systems [3]. We present our variant of this logic in detail and show some results that demonstrate its suitability for the specification of dynamically reconfigurable systems. In particular, we prove that the logic, with a special type of *language translation*, constitutes an *institution* [13]. This result enables us, if specifications are organised appropriately, to import properties from components when building (dynamic) amalgamations, as we will show. We discuss some shortcomings of STeP in supporting the required forms of reasoning and whether they can be overcome via encodings or whether they require an extension to STeP.

## 2 Some Requirements to Express Dynamic Reconfiguration

We motivate here the changes we propose to the original logic, necessary to deal with dynamic reconfiguration. We present a simple example where the reasons that motivate the changes to the logic become apparent. The style of specification, or modelling, we want to use is inspired by the one in [9] and related work.

### 2.1 Representing Action Occurrence

Consider a system where printers and print servers exist, with the corresponding interactions between them. A varying number of printers and servers is necessary, in order to provide flexibility and robustness. Printers have very simple behaviour. They can load a job, which is a string of characters, and they can print a previously loaded job. Standard printers do not have a buffer, though.

A possible way of modelling printers as components is the following: the *attributes* of a printer are its current job and one that indicates whether it is ready or not to receive a new job. The *operations* that a printer can perform are: *(i)* load, which allows a printer to load a new job, *(ii)* print, which prints the current job, *(iii)* print-el, which serves the purpose of implementing by successive calls the print operation, by printing the current character of the job to be printed, and *(iv)* p-init, which initialises a printer by setting the current job to the empty string and making the printer ready. So, a *template* modelling the structure of printers (only the structure, i.e., a kind of signature of printers) would look like the one in Figure 1.

**Class** *Printer*
**Attributes:** *ready* : boolean, *job* : string
**Actions:** *p-init*(), *print*(), *load*($j$ : string), *print-el*($c$ : char)
**EndofClass**

**Fig. 1.** Class *Printer*.

Of course, we need to indicate somehow the intention of actions, i.e., their effect on attributes. We opt to do so by using temporal logic, so components can be specified in a declarative way, and with direct support for reasoning about their properties. More specifically, we would like to use the Manna-Pnueli temporal logic, which has been successfully used to specify reactive systems. Clearly, we want an attribute's values to be state dependent. So, we can model them using what are called *local variables* in the nomenclature of the Manna-Pnueli logic, i.e., 0-ary flexible function symbols. We can model (instantaneous) actions of the printer component as predicate symbols, in such a way that the truth of $a(x)$ represents the occurrence of action $a$ with parameter $x$. Then, we would like some predicate symbols to have a state dependent interpretation. The flexibility of $a$ intuitively indicates that $a(x)$, for some $x$, can *happen* in some instants of time, and not happen in some other instants (as a static characterisation of $a$ would force $a(x)$ to happen either in all or none of the states of the system).

In the Manna-Pnueli logic though, all predicate symbols have a state independent interpretation. So, we cannot model actions straightforwardly, in the style described above. We need then to consider a variant of the Manna-Pnueli logic, one in which (at least) some predicate symbols are allowed to have a state dependent interpretation. The reader might argue that a similar effect can be achieved by specifying functions from the parameters of $a$ to booleans as a datatype $P$, and declaring $a$ to be a local variable of type $P$. However, we choose to consider a variant of the Manna-Pnueli logic instead, so the study of the logic and the combination of different languages and theories, which will be necessary in order to put together the specifications of different components, is easier to achieve. See section 5 for more details in this regard.

In the presence of "flexible" predicate symbols for describing the actions of components, we can provide temporal formulae to specify the above component's behaviour. For instance, the following formula could be part of a temporal specification for printers:

$$\Box[\forall j \in \mathsf{string} : \mathit{load}(j) \rightarrow \bigcirc(\mathit{job} = j)]$$

which, by means of the next and box operators, intuitively describes the post-condition of action $\mathit{load}(j : \mathsf{string})$.

## 2.2 Building Aggregations of Components

Suppose that in addition to printers, we also specify other necessary kinds of components, such as print servers and buffers. After that, we need to put them together to build (dynamically evolving) aggregations of components. Contrary to the way (static) aggregations are constructed in algebraic specifications and some formal component based formalisms, using categorical diagrams to represent composite systems, we do it by implementing in a logical way a concept similar to the "dot notation" of object orientation. In order to do so, we add special arguments to actions and attributes to denote the instances to which these belong. So, for instance, attributes *job* and *ready* from the specification of printers, which originally are of type $\mathsf{string}$ and $\mathsf{boolean}$ respectively, will become of type $N \rightarrow \mathsf{string}$ and $N \rightarrow \mathsf{boolean}$, for a sort $N$ representing the names of instances of printers.

Then, we are in a situation in which we need state dependent function symbols of arity greater than zero, in theories describing dynamic aggregations of components. As we previously observed, in the Manna-Pnueli logic only local variables (special function symbols of arity zero) are state dependent. So, we need to introduce another modification to the Manna-Pnueli logic. Again, the reader might argue that a similar effect can be achieved by defining suitable datatypes for these attributes, but, as for the case of flexible predicate symbols, this complicates the study of the logic and the way descriptions of different components can be combined. Again, we refer the reader to section 5 for a more detailed explanation.

## 3 A Logic for the Description of Reconfigurable Systems

We now start describing the logic we use as a core for the specification of reconfigurable systems. Earlier versions of it have been presented in [2][3], and as indicated above, this is a variant of a logic widely used for the specification of reactive systems, namely the Manna-Pnueli logic [16][17]. Some of the definitions in this section are adapted from others in [16] and [21]. The use of the logic for expressing properties of systems is standard. We will generalise the notion of local variables ("constants" whose values are state dependent) to flexible function symbols (whose interpretation is state dependent). Predicate symbols are also split between *rigid* and *flexible*. Thus, we go back to earlier work, such as [1], which led to the present status of the Manna-Pnueli logic, where flexible symbols were more general than simply 0-ary functions.

In contrast with previous work on the Manna-Pnueli logic, we focus on the use of (our variant of) the logic in the context of different alphabets and their transformations, i.e., we use it in an *institutional* way [13].

### 3.1 Syntax

An *alphabet* (sometimes called *signature* or *vocabulary*) for this logic consists of:

– a finite set $\mathbf{S}$ of sorts,
– a finite set of $(\mathbf{S}^* \times \mathbf{S})$-indexed flexible function symbols,
– a finite set of $(\mathbf{S}^* \times \mathbf{S})$-indexed rigid function symbols,
– a finite set of $(\mathbf{S}^*)$-indexed flexible predicate symbols,
– a finite set of $(\mathbf{S}^*)$-indexed rigid predicate symbols,
– a countable set of $\mathbf{S}$-indexed (logical) variables.

Typed terms are constructed from the symbols of the vocabulary in the standard way, without making any distinction between flexible and rigid function symbols. Formulae are constructed also in the usual way (without any distinction between flexible and rigid predicates), using the traditional propositional connectives, the unary temporal operators $\bigcirc$ and $\Diamond$, the binary temporal operator $\mathcal{U}$, and quantification over variables.

Terms are precisely defined in the following way:

**Definition 1.** *Given an alphabet $\mathcal{A}$, the set of typed* terms *for it is constructed as follows:*

- *if $V$ is a variable of sort $S$ in $\mathcal{A}$, then $V$ is a term of sort $S$,*
- *if $f : S_1, \ldots, S_k \times S$ is a (flexible or rigid) function symbol in $\mathcal{A}$, and $t_1, \ldots, t_k$ are terms of sorts $S_1, \ldots, S_k$ respectively, then $f(t_1, \ldots, t_k)$ is a term of sort $S$,*
- *if $t$ is a term of sort $S$, so is $\bigcirc t$,*
- *no other string of symbols is a term for $\mathcal{A}$.*

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. From now on, we will use "$\rightarrow$" instead of "$\times$" for indicating the type of function symbols, since it has a more intuitive reading in our context.

Formulae, used to build assertions about individuals, are constructed as follows:

**Definition 2.** *Given an alphabet $\mathcal{A}$, the set of* formulae *for it is constructed as follows:*

- *if $t_1$ and $t_2$ are terms of sort $S$ for $\mathcal{A}$, then $t_1 = t_2$ is a formula,*
- *if $t_1, \ldots, t_k$ are terms of sorts $S_1, \ldots, S_k$ respectively, and $p : S_1, \ldots, S_k$ is a (flexible or rigid) predicate symbol, then $p(t_1, \ldots, t_k)$ is a formula, called* atomic*,*
- *if $\alpha$ and $\beta$ are formulae, so are $\neg\alpha, \alpha \rightarrow \beta, \bigcirc\alpha, \Diamond\alpha, \alpha\mathcal{U}\beta$,*
- *if $V$ is a variable of sort $S$ and $\alpha$ is a formula, then $\forall V \in S : \alpha$ is a formula,*
- *no other string of symbols is a formula for $\mathcal{A}$.*

*We denote by $L_{\mathcal{A}}$ the set of formulae over an alphabet $\mathcal{A}$.*

The intended meaning of propositional connectives is 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 "$\alpha$ is true in the next state" and "$\alpha$ is true (at least) until $\beta$ becomes true", respectively. This is formalised in the section about the semantics of this logic. The box operator, that we used in the previous section, is defined in terms of the diamond operator and negation, as:

$$\Box\alpha \equiv \neg\Diamond\neg\alpha.$$

It has its own intuitive reading: $\Box\alpha$ intuitively means "always in the future, $\alpha$ holds", with a reflexive semantics for "always in the future", i.e., the "future" includes the "present".

## 3.2 Semantics

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

**Definition 3.** *Given an alphabet $\mathcal{A}$, a (semantic) structure $M$ for it is a mapping that assigns:*

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

$$f^M : S_1^M, \ldots, S_k^M \to S^M,$$

- *for each rigid predicate symbol $p : S_1, \ldots, S_k$ in $\mathcal{A}$, 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 $f : S_1, \ldots, S_k \to S$ in $\mathcal{A}$ to a function

$$f^M : S_1^M, \ldots, S_k^M \to S^M,$$

- every flexible predicate symbol $p : S_1, \ldots, S_k$ in $\mathcal{A}$ to a relation

$$p^M \subseteq S_1^M \times \ldots \times S_k^M.$$

A *trajectory* $\sigma$ 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 $A_{x/d}$ the assignment that coincides with $A$ for all variables $y \neq x$, and that maps $x$ to $d$.

**Definition 4.** *Let $\mathcal{A}$ be an alphabet. An* interpretation *is a triple $I = (M, A, \sigma)$, where $M$ is an $\mathcal{A}$-structure, $A$ is an assignment for $M$, and $\sigma$ is a trajectory in $M$.*

Given an interpretation $I = (M, A, \sigma)$, we denote by $I_{x/d}$ the interpretation $(M, A_{x/d}, \sigma)$.

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

- for a variable $v$, $I(v) = A(v)$,
- for a rigid 0-ary function symbol $f :\to S$, $I(f) = f^M$,
- for a flexible 0-ary function symbol $f :\to S$, $I(f) = \sigma_0(f)$, where $\sigma_0$ is the first state in the trajectory $\sigma$,
- 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,

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

– for a term $\bigcirc t$, $I(\bigcirc t) = I^{(1)}(t)$, where $I^{(1)} = (M, A, \sigma^{(1)})$

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

**Definition 5.** *Let $\mathcal{A}$ be an alphabet, and $I = (M, A, \sigma)$ an interpretation. We define satisfaction of a formula $\alpha$ (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* $I(t_1) = I(t_2)$,
– $\overset{I}{\models} p(t_1, \ldots, t_k)$, *where $p$ is a rigid predicate symbol, if and only if*

$$(I(t_1), \ldots, I(t_k)) \in p^M,$$

– $\overset{I}{\models} p(t_1, \ldots, t_k)$, *where $p$ is a flexible predicate symbol, if and only if*

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

– $\overset{I}{\models} \neg\beta$ *if and only if is not the case that* $\overset{I}{\models} \beta$,
– $\overset{I}{\models} \beta_1 \rightarrow \beta_2$ *if and only if either* $\overset{I}{\models} \neg\beta_1$ *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* $0 \leq i < k$, $\overset{I^{(i)}}{\models} \beta_1$,
– $\overset{I}{\models} \forall x \in S : \beta$ *if and only if for all* $d \in S^M$ *it is the case that* $\overset{I_{x/d}}{\models} \beta$.

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

**Semantic Consequence** We overload the symbol $\models$, using it now for defining a relation between sets of formulae over an alphabet $\mathcal{A}$.

**Definition 6.** *Let $\mathcal{A}$ be an alphabet, $\Phi$ and $\Psi$ sets of formulae over $\mathcal{A}$. Then, we say that $\Psi$ is a* semantic consequence *of $\Phi$, in symbols $\Phi \models \Psi$, if and only if for every interpretation $I$ (for $\mathcal{A}$), if $I$ is a model of $\Phi$ then it is also a model of $\Psi$.*

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

- *it is* reflexive*, i.e., for every set of formulae $\Phi$, it is the case that $\Phi \models \Phi$,*
- *it satisfies* cut*:*

  *For all sets $\Phi$, $\Phi_1$ and $\Psi$ of formulae, if $\Phi \cup \Phi_1 \models \Psi$ and $\Phi \models \phi$, for all $\phi$ in $\Phi_1$, then $\Phi \models \Psi$,*
- *it is* monotonic*, i.e., if $\Phi$ and $\Psi$ are sets of formulae such that $\Phi \models \Psi$, then $\Phi \cup \Phi_1 \models \Psi$, for every set $\Phi_1$ of formulae.*

**Structurality of Semantic Consequence** It will be necessary for us to combine different languages and formulae in order to be able to build specifications. For this purpose, we need to prove that the logic satisfies certain structural properties.

**Definition 7.** *We define an* alphabet morphism *$\tau$ between alphabets*

$$\mathcal{A} = (\mathbf{S}_{\mathcal{A}}, \mathbf{F}^{fl}_{\mathcal{A}}, \mathbf{F}^{rg}_{\mathcal{A}}, \mathbf{P}^{fl}_{\mathcal{A}}, \mathbf{P}^{rg}_{\mathcal{A}}, \mathbf{V}_{\mathcal{A}})$$

*and*

$$\mathcal{B} = (\mathbf{S}_{\mathcal{B}}, \mathbf{F}^{fl}_{\mathcal{B}}, \mathbf{F}^{rg}_{\mathcal{B}}, \mathbf{P}^{fl}_{\mathcal{B}}, \mathbf{P}^{rg}_{\mathcal{B}}, \mathbf{V}_{\mathcal{B}})$$

*as a family of functions:*

- *$\tau_{sorts}$, mapping each sort in $\mathbf{S}_{\mathcal{A}}$ to a sort in $\mathbf{S}_{\mathcal{B}}$,*
- *$\tau_{fl\text{-}fnt}$, mapping each flexible function symbol*

$$f : S_1, \ldots, S_k \rightarrow S$$

  *in $\mathbf{F}^{fl}_{\mathcal{A}}$ to a flexible function symbol*

$$\tau_{fl\text{-}fnt}(f) : \tau_{sorts}(S_1), \ldots, \tau_{sorts}(S_k), S'_{k+1}, \ldots, S'_n \rightarrow \tau_{sorts}(S)$$

  *in $\mathbf{F}^{fl}_{\mathcal{B}}$, where $S'_{k+1}, \ldots, S'_n$ do not belong to the image of $\tau_{sorts}$,*
- *$\tau_{rg\text{-}fnt}$, mapping each rigid function symbol*

$$f : S_1, \ldots, S_k \rightarrow S$$

  *in $\mathbf{F}^{rg}_{\mathcal{A}}$ to a rigid function symbol*

$$\tau_{rg\text{-}fnt}(f) : \tau_{sorts}(S_1), \ldots, \tau_{sorts}(S_k), S'_{k+1}, \ldots, S'_n \rightarrow \tau_{sorts}(S)$$

  *in $\mathbf{F}^{rg}_{\mathcal{B}}$, where $S'_{k+1}, \ldots, S'_n$ do not belong to the image of $\tau_{sorts}$,*
- *$\tau_{fl\text{-}prd}$, mapping each flexible predicate symbol*

$$p : S_1, \ldots, S_k$$

  *in $\mathbf{P}^{fl}_{\mathcal{A}}$ to a flexible predicate symbol*

$$\tau_{fl\text{-}prd}(p) : \tau_{sorts}(S_1), \ldots, \tau_{sorts}(S_k), S'_{k+1}, \ldots, S'_n$$

  *in $\mathbf{P}^{fl}_{\mathcal{B}}$, where $S'_{k+1}, \ldots, S'_n$ do not belong to the image of $\tau_{sorts}$,*

– $\tau_{rg\text{-}prd}$, *mapping each rigid predicate symbol*

$$p : S_1, \ldots, S_k$$

*in* $\mathbf{P}^{rg}_{\mathcal{A}}$ *to a rigid predicate symbol*

$$\tau_{rg\text{-}prd}(p) : \tau_{sorts}(S_1), \ldots, \tau_{sorts}(S_k), S'_{k+1}, \ldots, S'_n$$

*in* $\mathbf{P}^{rg}_{\mathcal{B}}$*, where* $S'_{k+1}, \ldots, S'_n$ *do not belong to the image of* $\tau_{sorts}$*,*

– $\tau_{vars}$*, mapping each variable* $x : S$ *in* $\mathbf{V}_{\mathcal{A}}$ *to a variable* $\tau_{vars}(x) : \tau_{sorts}(S)$ *in* $\mathbf{V}_{\mathcal{B}}$*.*

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.

In the following definition, we denote the image set of a function $f$ by $Img(f)$.

**Definition 8.** *Let* $\tau : \mathcal{A} \to \mathcal{B}$ *be an alphabet morphism. We define the function*

$$Gr_\tau : L_{\mathcal{A}} \to L_{\mathcal{B}}$$

*as follows: Given a formula* $\alpha$*, the formula* $Gr_\tau(\alpha)$ *results from the following procedure:*

1. *We first (inductively) apply the translation of the symbols in* $\alpha$ *using* $\tau$*. Let us call this pseudo-formula* $\alpha_\tau$*.*
2. *For every sort in* $\mathcal{B} - Img(\tau_{sorts})$*, we choose a variable. If the translation of* $\alpha$ *into* $\alpha_\tau$ *involved translating function or predicate symbols to others of a greater arity (recall that the definition of alphabet morphism allows for this), then the remaining arguments are filled with these chosen variables, of the corresponding sort. (Note that all the remaining arguments of function symbols or predicates of a specific sort will be filled with the same variable.)*
3. *We universally quantify (over the corresponding sorts) all the free new variables added as a result of the previous step.*

*Example 1.* Consider the following formula $\alpha$:

$$\Box[(\exists x \in S : p(x)) \to q].$$

Let $\tau$ be an alphabet morphism mapping $S$ to $S'$, $p : S$ to $p' : S', S''$, $q$ to $q' : S''$, and $x : S$ to $x : S'$. Then, the formula $Gr_\tau(\alpha)$ resulting from the translation of $\alpha$ is:

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

**Definition 9.** *Let $\tau : \mathcal{A} \to \mathcal{B}$ be an alphabet morphism and $I = (M, A, \sigma)$ a $\mathcal{B}$-interpretation. Let $S_{\mathcal{B}_1}, \ldots, S_{\mathcal{B}_j}$ be the sorts in $\mathcal{B} - Img(\tau_{sorts})$, and $d_1, \ldots, d_j$ elements in $S^M_{\mathcal{B}_1}, \ldots, S^M_{\mathcal{B}_j}$, respectively.*

*We denote by $I_{|_{\mathcal{A}}^{d_1,\ldots,d_j}}$ the $\mathcal{A}$-interpretation $\left(M_{|_{\mathcal{A}}^{d_1,\ldots,d_j}}, A_{|_{\mathcal{A}}^{d_1,\ldots,d_j}}, \sigma_{|_{\mathcal{A}}^{d_1,\ldots,d_j}}\right)$, where:*

- *$M_{|_{\mathcal{A}}^{d_1,\ldots,d_j}}$ is defined as:*
  - *Each sort $S$ in $\mathcal{A}$ is interpreted as $(\tau_{sorts}(S))^M$,*
  - *each rigid function symbol $f$ in $\mathcal{A}$, which is translated via $\tau$ to a symbol of the same arity, is interpreted as $(\tau_{rg\text{-}fnt}(f))^M$,*
  - *each rigid function symbol $f : S_1, \ldots, S_k \to S$ in $\mathcal{A}$, which is translated via $\tau$ to a symbol of a greater arity*

    $$\tau_{rg\text{-}fnt}(f) : \tau_{sorts}(S_1), \ldots, \tau_{sorts}(S_k), S_{\mathcal{B}_{\pi(1)}}, \ldots, S_{\mathcal{B}_{\pi(n)}} \to \tau_{sorts}(S),$$

    *is interpreted as the function:*

    $$(\tau_{rg\text{-}fnt}(f))^M_{|_{\mathcal{A}}^{d_1,\ldots,d_j}} : (\tau_{sorts}(S_1))^M, \ldots, (\tau_{sorts}(S_k))^M \to (\tau_{sorts}(S))^M,$$

    *defined as:*

    $$(\tau_{rg\text{-}fnt}(f))^M_{|_{\mathcal{A}}^{d_1,\ldots,d_j}}(x_1, \ldots, x_k) = \\ (\tau_{rg\text{-}fnt}(f))^M(x_1, \ldots, x_k, d_{\pi(1)}, \ldots, d_{\pi(n)}),$$

  - *each rigid predicate symbol $p$ in $\mathcal{A}$, which is translated via $\tau$ to a symbol of the same arity, is interpreted as $(\tau_{rg\text{-}prd}(p))^M$,*
  - *each rigid predicate symbol $p : S_1, \ldots, S_k$ in $\mathcal{A}$, which is translated via $\tau$ to a symbol of a greater arity*

    $$\tau_{rg\text{-}prd}(p) : \tau_{sorts}(S_1), \ldots, \tau_{sorts}(S_k), S_{\mathcal{B}_{\pi(1)}}, \ldots, S_{\mathcal{B}_{\pi(n)}}$$

    *is interpreted as the relation:*

    $$(\tau_{rg\text{-}prd}(p))^M_{|_{\mathcal{A}}^{d_1,\ldots,d_j}} \subseteq (\tau_{sorts}(S_1))^M \times \ldots \times (\tau_{sorts}(S_k))^M,$$

    *defined as:*

    $$\{(x_1, \ldots, x_k) | (x_1, \ldots, x_k, d_{\pi(1)}, \ldots, d_{\pi(n)}) \in (\tau_{rg\text{-}prd}(p))^M\}$$

  *where $\pi$ is a permutation of $1, \ldots, j$, (and $S_{\mathcal{B}_{\pi(1)}}, \ldots, S_{\mathcal{B}_{\pi(n)}}$ represents a subsequence of a permutation of $S_{\mathcal{B}_1}, \ldots, S_{\mathcal{B}_j}$)*
- *$A_{|_{\mathcal{A}}^{d_1,\ldots,d_j}}$ maps each variable $x$ is $\mathcal{A}$ to $A(\tau_{vars}(x))$,*
- *each state $s_i$ of the trajectory $\sigma_{|_{\mathcal{A}}^{d_1,\ldots,d_j}}$ is defined as:*
  - *each flexible function symbol $f$ in $\mathcal{A}$, which is translated via $\tau$ to a symbol of the same arity, is interpreted as $s_i^I(\tau_{fl\text{-}fnt}(f))$,*

- *each flexible function symbol $f : S_1, \ldots, S_k \to S$ in $\mathcal{A}$, which is translated via $\tau$ to a symbol of a greater arity*
$$\tau_{\mathit{fl\text{-}fnt}}(f) : \tau_{\mathit{sorts}}(S_1), \ldots, \tau_{\mathit{sorts}}(S_k), S_{\mathcal{B}_{\pi(1)}}, \ldots, S_{\mathcal{B}_{\pi(n)}} \to \tau_{\mathit{sorts}}(S),$$
*is interpreted as the function:*
$$s_i(f) : (\tau_{\mathit{sorts}}(S_1))^M, \ldots, (\tau_{\mathit{sorts}}(S_k))^M \to (\tau_{\mathit{sorts}}(S))^M,$$
*defined as:*
$$(s_i(f))(x_1, \ldots, x_k) = (s_i^I(\tau_{\mathit{fl\text{-}fnt}}(f)))(x_1, \ldots, x_k, d_{\pi(1)}, \ldots, d_{\pi(n)}),$$
- *each flexible predicate symbol $p$ in $\mathcal{A}$, which is translated via $\tau$ to a symbol of the same arity, is interpreted as $s_i^I(\tau_{\mathit{fl\text{-}prd}}(p))$,*
- *each flexible predicate symbol $p : S_1, \ldots, S_k$ in $\mathcal{A}$, which is translated via $\tau$ to a symbol of a greater arity*
$$\tau_{\mathit{fl\text{-}prd}}(p) : \tau_{\mathit{sorts}}(S_1), \ldots, \tau_{\mathit{sorts}}(S_k), S_{calB_{\pi(1)}}, \ldots, S_{\mathcal{B}_{\pi(n)}}$$
*is interpreted as the relation:*
$$s_i(p) \subseteq (\tau_{\mathit{sorts}}(S_1))^M \times \ldots \times (\tau_{\mathit{sorts}}(S_k))^M,$$
*defined as:*
$$\{(x_1, \ldots, x_k) | (x_1, \ldots, x_k, d_{\pi(1)}, \ldots, d_{\pi(n)}) \in s_i^I(\tau_{\mathit{fl\text{-}prd}}(p))\},$$

*where $s_i^I$ is the i-th state in the trajectory $\sigma$, and $\pi$ is a permutation of $1, \ldots, j$.*

**Lemma 1.** *Let $\tau : \mathcal{A} \to \mathcal{B}$ be an alphabet morphism, $\phi$ an $\mathcal{A}$-formula and $I = (M, A, \sigma)$ a $\mathcal{B}$-interpretation. Let $S_{\mathcal{B}_1}, \ldots, S_{\mathcal{B}_j}$ be the sorts in $\mathcal{B}-\mathit{Img}(\tau_{\mathit{sorts}})$, and $d_1, \ldots, d_j$ arbitrary elements in $(S_{\mathcal{B}_1})^M, \ldots, (S_{\mathcal{B}_j})^M$, respectively. Then,*
$$I_{x_1, \ldots, x_k / d_{\pi_i(1)}, \ldots, d_{\pi_i(k)}}$$
*is a model of $\phi_\tau(x_1, x_2, \ldots, x_k)$ if and only if*
$$I_{|\mathcal{A}}^{d_{\pi_i(1)}, \ldots, d_{\pi_i(k)}}$$
*is a model of $\phi$.*

*Proof.* The proof proceeds by induction over the structure of formula $\phi$. It is relatively straightforward, and is left as an exercise for the interested reader.

**Lemma 2.** *Let $\tau : \mathcal{A} \to \mathcal{B}$ be an alphabet morphism, $\Phi$ a set of $\mathcal{A}$-formulae and $I$ a $\mathcal{B}$-interpretation. Let $S_{\mathcal{B}_1}, \ldots, S_{\mathcal{B}_j}$ be the sorts in $\mathcal{B} - Img(\tau_{sorts})$, and $d_1, \ldots, d_j$ arbitrary elements in $M(S_{\mathcal{B}_1}), \ldots, M(S_{\mathcal{B}_j})$, respectively. Then, if $I$ is a model of $Gr_\tau(\Phi)$, $I_{|_{\mathcal{A}}}^{d_1,\ldots,d_j}$ is an $\mathcal{A}$-model of $\Phi$.*

*Proof.* Let $\tau : \mathcal{A} \to \mathcal{B}$ be an alphabet morphism, $\Phi$ a set of $\mathcal{A}$-formulae and $I$ a $\mathcal{B}$-interpretation, which is a model of $Gr_\tau(\Phi)$.

First, note that for each formula $\phi_i$ in $\Phi$, its translation $Gr_\tau(\phi_i)$ in $Gr_\tau(\Phi)$ is of the form:

$$\forall x_1 \in S'_1 : \forall x_2 \in S'_2 ... \forall x_k \in S'_k : \phi_{i_\tau}(x_1, x_2, \ldots, x_k),$$

where $\phi_{i_\tau}$ is the pseudo-formula obtained from $\phi_i$ after the first step of the transformation, and $\phi_{i_\tau}(x_1, x_2, \ldots, x_k)$ is the one obtained after the second step of the transformation, i.e., with the "empty places" appropriately filled with $x_1, x_2, \ldots, x_k$. Due to the definition of the transformation $Gr_\tau$, it is clear that $S'_1, \ldots, S'_k$ are not in $Img(\tau_{sorts})$. Let us assume that

$$\mathcal{B} - Img(\tau_{sorts}) = \{S_{\mathcal{B}_1}, \ldots S_{\mathcal{B}_j}\}.$$

Then, we can rewrite each $Gr_\tau(\phi_i)$ as:

$$\forall x_1 \in S_{\mathcal{B}_{\pi_i(1)}} : \forall x_2 \in S_{\mathcal{B}_{\pi_i(2)}} ... \forall x_k \in S_{\mathcal{B}_{\pi_i(k)}} : \phi_{i_\tau}(x_1, x_2, \ldots, x_k)$$

where $\pi_i$ denotes a permutation.

Variables $x_1, x_2, \ldots, x_k$ always occur free in $\phi_{i_\tau}(x_1, x_2, \ldots, x_k)$. This is due to the fact that they are typed with sorts not used by the first step of the translation $Gr_\tau$. Also, they are different from each other, due to the fact that they are typed with different sorts.

Since $I$ is a model of $Gr_\tau(\Phi)$, it is a model of each of the $\phi'_{i_\tau}$. Then, for any possible assignment, mapping $x_1, x_2, \ldots, x_k$ to elements $v_1, v_2, \ldots, v_k$ in $I(S_{\mathcal{B}_{\pi_i(1)}}), \ldots, I(S_{\mathcal{B}_{\pi_i(k)}})$, respectively, we know that

$$\overset{I_{x_1,\ldots,x_k/v_1,\ldots,v_k}}{\models} \quad \phi_{i_\tau}(x_1, x_2, \ldots, x_k).$$

Let $d_1, \ldots, d_j$ be arbitrary elements in $I(S_{\mathcal{B}_{\pi_i(1)}}), \ldots, I(S_{\mathcal{B}_{\pi_i(k)}})$. Let us denote by $A_1$ the assignment mapping all variables of sort $S_{i_\mathcal{B}}$ to the value $d_i$. Again, under this assignment, clearly the following holds:

$$\overset{I_{x_1,\ldots,x_k/d_{\pi_i(1)},\ldots,d_{\pi_i(k)}}}{\models} \quad \phi_{i_\tau}(x_1, x_2, \ldots, x_k).$$

Therefore, due to our previous Lemma, $I_{|_{\mathcal{A}}}^{d_{\pi_i(1)},\ldots,d_{\pi_i(k)}}$ is a model of each $\phi_i$, i.e., it is a model of $\Phi$, as we wanted to prove.

**Theorem 1.** *Let $\tau : \mathcal{A} \to \mathcal{B}$ be an alphabet morphism, and $\Phi$ and $\Psi$ be sets of $\mathcal{A}$-formulae such that $\Phi \models \Psi$. Then, $Gr_\tau(\Phi) \models Gr_\tau(\Psi)$.*

*Proof.* Let $\tau : \mathcal{A} \rightarrow \mathcal{B}$ be an alphabet morphism, and $\Phi$ and $\Psi$ be sets of $\mathcal{A}$-formulae such that $\Phi \models \Psi$. Let $I$ be a model of $Gr_\tau(\Phi)$. Let $S_{\mathcal{B}_1}, \ldots, S_{\mathcal{B}_j}$ be the sorts in $\mathcal{B} - Img(\tau_{sorts})$. According to the previous Lemma, for every set $d_1, \ldots, d_j$ of elements in $I(S_{\mathcal{B}_1}), \ldots, I(S_{\mathcal{B}_j})$, we have that $I^{d_1,\ldots,d_j}_{|_{\mathcal{A}}}$ is a model of $\Phi$. Then, due to our hypothesis, $I^{d_1,\ldots,d_j}_{|_{\mathcal{A}}}$ is also a model of $\Psi$. Therefore, due to the first of our Lemmas, $I_{x_1,\ldots,x_k/d_1,\ldots,d_k}$ is a model of each of the $\psi_{\tau_i}$ in $Gr_\tau(\Psi)$. Then, due to the definition of satisfaction for quantification, $I$ is a model of each $Gr_\tau(\Psi)$, as we wanted to prove.

**A Semantic $\pi$-Institution** We present here a result that is crucial for our application of the presented formalism. The result is introduced by using a particular kind of institution, the $\pi$-institution. Institutions [13] are a formalisation of the notion of logical system, which have been proposed as a way of dealing with logical systems at a greater level of abstraction. Institutions embed notions of model and satisfaction, together with signatures and sentences. $\pi$-institutions [10], on the other hand, concentrate on the notion of consequence, without considering model-theoretic issues. Although we have defined a notion of model in our logic, we just need to deal with (semantic) consequence at this stage. So, $\pi$-institutions suffice for our needs. We reproduce below the formal definition of $\pi$-institutions. We refer the reader to [13] for more details regarding institutions (and their applications), in general, and to [10] for $\pi$-institutions, in particular.

**Definition 10.** *A* $\pi$-institution *is a triple* $(Sig, Gram, \vdash)$*, where:*

- *Sig is a category [5] (of signatures or alphabets),*
- *Gram* $: Sig \rightarrow Set$ *is a (grammar) functor,*
- $\vdash$ *is a Sig-indexed family of consequence relations, each of which is a relation in* $\wp(Gram(\Sigma)) \times Gram(\Sigma)$ *for the corresponding alphabet* $\Sigma$*, such that for every* $\phi \in Gram(\Sigma)$*,* $\Phi, \Psi \in \wp(Gram(\Sigma))$*, the following properties are satisfied:*
  - reflexivity*:* $\Phi \vdash \phi$*, if* $\phi \in \Phi$*,*
  - cut*: if* $\Phi \cup \Psi \vdash \phi$ *and* $\Phi \vdash \psi$*, for all* $\psi \in \Psi$*, then* $\Phi \vdash \phi$*,*
  - monotonicity*: if* $\Phi \vdash \phi$ *then* $\Phi \cup \Psi \vdash \phi$*,*
  - structurality*: for every* $\tau : \Sigma \rightarrow \Sigma'$*, if* $\Phi \vdash \phi$ *then* $Gram(\tau)(\Phi) \vdash Gram(\tau)(\phi)$*.*

The following result indicates that the alphabet translations of our logic preserve semantic consequence.

**Theorem 2.** *The structure* $(Sig, Gr, \models)$*, where*

- *Sig is the set-theoretic category, where alphabets are the objects and alphabet morphisms are the arrows,*
- *Gr is a functor from Sig to the category Set, that maps each alphabet* $\mathcal{A}$ *to the set of* $\mathcal{A}$*-formulae, and each alphabet morphism* $\tau : \mathcal{A} \rightarrow \mathcal{B}$ *to the function* $Gr_\tau : L_{\mathcal{A}} \rightarrow L_{\mathcal{B}}$*,*
- $\models$ *is the semantic consequence relation of the logic,*

*is a* $\pi$-institution*.*

*Proof.* The proof is straightforward. It follows directly from the definition of $\pi$-institutions [10], and the previously proved Theorem 1 and Proposition 1.

## 4 Building Specifications of Systems

We now explain how the above results can be exploited to effectively specify dynamically reconfigurable component based systems. Specifications will be modularly organised in layers, from datatype specifications to the specification of architectural subsystems.

### 4.1 Basic Datatypes

At the core of the language is the definition of basic abstract datatypes. Basic components will build their state by means of variables, as in imperative programming languages, of types defined in the abstract datatypes specification. It is not difficult to see that the logic has sufficient expressive power to deal with datatype specifications in the style of algebraic specifications [7], since the logic is first-order.

A datatype specification is simply a *static* theory presentation, i.e., a theory presentation where the sets of flexible predicates and flexible functions are empty.

Let us assume that we have such a theory presentation, which contains the definition of all standard datatypes, such as integers, sequences, natural numbers, etc. Let us denote this theory presentation by:

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

### 4.2 Basic Components

Components are one of the basic building blocks of software architectures. As we previously described, we intend to use temporal logic theories to describe such components, as in [9]. Since a varying number of "instances" of the same type of component can be held in the systems we are interested in, we want a way of describing templates of these components, instead of the components themselves. We call these descriptions *class definitions.* Class definitions are modularly built on top of an underlying datatype specification, $\mathcal{ADT}$ in our case.

A class definition consists of: *(i)* a name, *(ii)* finite sets of attributes and read variables whose type is a sort defined in $\mathcal{ADT}$, *(iii)* a finite set of actions, which can have arguments typed with sorts in $\mathcal{ADT}$. Furthermore, a class specification is equipped with a set of formulae, its *axioms*, over the alphabet:

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

where $C$ is the name of the class, and $Att$, $Rv$ and $Act$ denote the sets of attributes, read variables and actions respectively. That is, we use read variables and attributes as flexible function symbols, and actions as flexible predicates, extending the vocabulary defined in the datatype specification[1]. The name of the class is also used, as a 0-ary flexible predicate symbol.

The purpose of the axioms of a class specification is to describe the intended behaviour of the instances of the class, in a property oriented fashion. The flexible predicate symbols naming actions are used to represent action occurrence, as we explained before. The flexible function symbols naming attributes or read variables are used to represent the evolving values of these variables. The flexible predicate $C$, named after the name of the class, is used to represent the *activeness* of the corresponding object or instance. That is, the truth of $C$ in a particular state in a model represents the fact that the corresponding instance is active in that state. Note that $C$ represents a certain kind of structural information of the system. However, this is all the knowledge that a component can have, regarding the structure of the system. It is useful to have this kind of information, since usually one requires an instance to have a property *only* while it is active. Figure 2 is an axiomatisation for the printer class given in Figure 1.

1. $\Box[Printer \wedge p\text{-}init() \rightarrow \bigcirc(job = [])]$
2. $\Box[Printer \wedge p\text{-}init() \rightarrow \bigcirc(\neg p\text{-}init()\,\mathcal{U}\neg Printer)]$
3. $\Box[\forall j \in \mathsf{string} : Printer \wedge load(j) \rightarrow ready = \mathsf{T}]$
4. $\Box[\forall j \in \mathsf{string} : Printer \wedge load(j) \rightarrow \bigcirc(Printer \wedge (job = j))]$
5. $\Box[Printer \wedge print() \rightarrow (job \neq [])]$
6. $\Box[Printer \wedge print() \rightarrow \bigcirc job = \mathrm{tl}(job)]$
7. $\Box[Printer \wedge print() \rightarrow \bigcirc(Printer)]$
8. $\Box[Printer \wedge print() \rightarrow print\text{-}el(\mathrm{hd}(job))]$
9. $\Box[(ready = \mathsf{T}) \leftrightarrow (job = [])]$

**Fig. 2.** An Axiomatisation for Class *Printer*.

Axiom 2 illustrates the usefulness of predicate *Printer*. It indicates that once operation *p-init*() is called, while the instance is live, it cannot be called again during the "current lifetime" of the instance (note that a component might die and be created again later on).

A specification for a print server is slightly more complex, since a print server needs to send jobs to printers. However, we forbid the direct reference to other components, according to the above definition of class definitions. Communication can be achieved by action synchronisation and variable sharing, or even more complex interactions. The key point is that communication is completely externalised from class definitions. This forces the specifier to think about print servers as *closed* components. Logically, it allows us to reason about print servers

[1] Note that we maintain the "static" part of the abstract datatypes specification $\mathcal{ADT}$. In particular, we do not include new rigid function or predicate symbols. There is no technical reason for that, it is simply for the sake of simplicity. The specifier might want to define extra predicates, operations or even new datatypes for his or her component specification.

within a theory *independent* of other component definitions. In order to interact, a component accesses some information about the "environment" via special attributes called *read variables* (recall the definition of classes), which are not under the control of the component. Print servers use read variables.

For the case of print servers, we then have: *(i)* a queue of jobs as an attribute, *(ii)* a boolean read variable to indicate whether "the environment" is ready to receive a job, *(iii)* operations to enqueue a new job and to send a job to the "environment". Figure 3 contains a class definition for print servers.

**Class** *Server*
**Read Variables:** $p\text{-}ready : \mathsf{boolean}$
**Attributes:** $p\text{-}queue : \mathrm{list}(\mathsf{string})$
**Actions:** $s\text{-}init()$, $send()$, $enqueue(j : \mathsf{string})$, $dispatch(j : \mathsf{string})$
**EndofClass**

**Fig. 3.** Class *Server*.

An axiomatisation for print servers is shown in Figure 4.

1. $\Box[Server \wedge s\text{-}init() \rightarrow \bigcirc(p\text{-}queue = [])]$
2. $\Box[Server \wedge s\text{-}init() \rightarrow \bigcirc(\neg s\text{-}init()\,\mathcal{U}\neg Server)]$
3. $\Box[\forall j \in \mathsf{string} : Server \wedge enqueue(j) \rightarrow \bigcirc p\text{-}queue = p\text{-}queue +\!\!+ [j]]$
4. $\Box[Server \wedge send() \rightarrow (p\text{-}queue \neq [])]$
5. $\Box[Server \wedge send() \rightarrow dispatch(\mathrm{hd}(p\text{-}queue))]$
6. $\Box[Server \wedge send() \rightarrow \bigcirc p\text{-}queue = \mathrm{tl}(p\text{-}queue)]$
7. $\Box[(\exists j \in \mathsf{string} : dispatch(j)) \rightarrow send()]$
8. $\Box[\exists j_1, j_2 \in \mathsf{string} : dispatch(j_1) \wedge dispatch(j_2) \rightarrow (j_1 = j_2)]$

**Fig. 4.** An Axiomatisation for Class *Server*.

### 4.3 Building Dynamic Aggregations of Components

We have just described the way component types can be defined, and as we argued, we chose to define these class definitions as *closed* independent units. This allows us to reason about component properties independently of the rest of the system[2]. Now we want to start defining dynamic aggregations of components.

[2] Note that in traditional object oriented definitions, where there exists a notion of components represented by objects, this is not the case: classes are typically "defined" with explicit reference to other classes or other classes instances, by means of class-typed attributes. Methodologically, the "externalisation" of component interactions has several benefits, exploited by techniques such as those based on certain design patterns and the middleware technology.

But of course we need ways of making components interact. With that in mind, we declare a number of association definitions, whose meaning we will define later on. For the time being, associations have simply a name and a set of participants, typed with class names.

For our case study of printers and print servers, we might need an association *Prints-for* between print servers and printers. Figure 5 describes such an association.

**Association** *Prints-for*
**Participants:** $s$ : *Server*, $p$ : *Printer*
**EndofAssociation**

**Fig. 5.** *Prints-for*: An association definition relating print servers and printers.

Let us assume that the defined classes are $C_1, \ldots, C_k$ and the defined associations are $R_1, \ldots, R_m$. In order to build aggregations of components typed with the above classes and related by the above associations, we need to introduce an extra basic datatype. This extra datatype will represent names of instances and will help us characterise a concept similar to the "dot notation" of object orientation. Let $\mathcal{ADT}_{\mathsf{NAME}}$ be a static theory presentation, conservatively extending $\mathcal{ADT}$ with an extra sort $\mathsf{NAME}$, and a sufficiently large set of constants of this sort.

A *subsystem definition* is the definition of a complex component, whose internal structure is built out of the dynamic aggregation of interacting basic components. A subsystem is composed of: *(i)* a name, *(ii)* finite sets of basic attributes and basic read variables, typed by a sort in the alphabet of $\mathcal{ADT}_{\mathsf{NAME}}$, *(iii)* a finite set of operations, whose arguments are typed by sorts in the alphabet 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 *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:

- the sorts defined in $\mathcal{ADT}_{\mathsf{NAME}}$,
- the rigid function and predicate symbols defined in $\mathcal{ADT}_{\mathsf{NAME}}$,
- the flexible function and predicate symbols resulting from class definitions, adding to all of them an extra parameter of sort $\mathsf{NAME}$[3],

[3] We require the set of symbols of class definitions to be disjoint, in order to univocally determine the class a symbol belongs to. This is just to make the presentation

– 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 $a(x_1 : S_1, \ldots, x_k : S_k)$.

To better illustrate the intuition behind the constituents of a subsystem specification, let us consider an example.

**Subsystem** *Multiple_Printers*
**Attributes:** $s$ : NAME
**Operations:** *init*(), *change*($x$ : NAME), *add-p*($x$ : NAME), *del-p*($x$ : NAME)
**EndofSubsystem**

**Fig. 6.** A subsystem specification.

*Example 2.* With printers and print servers defined, we now want a dynamic subsystem consisting, at any point in time, of:

– a varying number of printers,
– one single print server, which can be dynamically replaced,
– an active printer, to which the print server is connected,
– operations for creating and deleting printers, for replacing print servers, and for changing the active printer of the subsystem.

Initially, this subsystem has only one live printer. We have a number of other restrictions. For instance, we do not want the subsystem to overload, so we allow for a maximum of 10 jobs in the queue of the contained print server.

Figure 6 defines a signature for such a subsystem. For the sake of simplicity, we consider a subsystem without read variables.

We can provide axioms to specify the operations, and the behaviour of the subsystem. We can start by describing the initial state:

$$\begin{aligned}&\mathit{Server}(s) \wedge \mathit{s\text{-}init}(s) \wedge \mathit{Printer}(p) \wedge \mathit{p\text{-}init}(p),\\ &\mathit{Prints\text{-}for}(s, p),\\ &\forall x : \mathit{Printer}(x) \rightarrow (x = p).\end{aligned}$$

The first of these axioms indicates that $s$ is a live server in the initial state, and that its initialisation operation is "called". The name $p$ is the one chosen for the initial printer, which is also initialised. The second formula says that server $s$ is initially connected to $p$ via *Prints-for*. Finally, the last of these axioms indicates that $p$ is the *only* live printer in the initial state. Note that we omit

simpler. A mechanism to individualise symbols of different classes can be easily implemented.

the sort when quantifying variables of sort $\mathsf{NAME}$, to make expressions more concise.

As can be seen, the flexible predicates named after the class names, which are incorporated from the language of the corresponding class definitions (with a relativisation represented by the extra argument), are useful for representing architectural information in the subsystem's axioms. In a similar fashion, the newly introduced flexible predicates named after association definitions are used to characterise the existence of connectors relating instances of classes.

Let us continue describing the subsystem. We have structural conditions: $s$ always references a live server, and is the reference to the only live print server:

$$\begin{array}{l}\Box[\mathit{Server}(s)],\\ \Box[\forall x : \mathit{Server}(x) \rightarrow (x = s)].\end{array}$$

Moreover, the queue of $s$ should never hold more than ten jobs (recall this is a restriction of the application)[4]:

$$\Box[\text{length}(\mathit{p\text{-}queue}(s)) \leq 10].$$

We can also describe the purpose of the operations. The following axioms describe the behaviour of the *change* operation:

$$\begin{array}{l}\Box[\forall x : \mathit{change}(x) \rightarrow \neg \mathit{Server}(x)],\\ \Box[\forall x, y : \mathit{change}(x) \wedge (s = y) \rightarrow \bigcirc(\mathit{s\text{-}init}(x)) \wedge \bigcirc(\neg \mathit{Server}(y)) \wedge (\bigcirc s = x)].\end{array}$$

The above example gives an idea of the way temporal formulae are used to characterise subsystem operations. The introduction of the signatures of basic components using the relativisation of their constituents is crucial. Basically, if *at* is an attribute defined in a class definition $C$, then symbol $\mathit{at}(n)$ represents attribute *at* for the instance named $n$. Similarly, if action $a(t_1, \ldots, t_i)$ is declared in a class definition $C$, then $a(t_1, \ldots, t_i, n)$ represents the occurrence of action $a$ with parameters $t_1, \ldots, t_i$ in the instance referred to as $n$. We prefer to use the "dot notation" for writing the extra parameter of attributes, read variables and actions introduced by the relativisation, to have more readable expressions. For instance, the expression

$$\Box[\text{length}(s.\mathit{p\text{-}queue}) \leq 10]$$

is a more readable way of writing one of the above axioms.

### 4.4 Describing Interactions

We previously introduced some flexible predicate symbols to talk about interactions among components. For the particular case of our example, we used a predicate *Prints-for*, to relate print servers and printers. We need to provide associations with meaning. We do so by means of corresponding formulae. Each

[4] We assume that $\leq$ is part of the datatype specification.

association definition is equipped with a finite set of formulae, over the signature of an including subsystem. There are some syntactic restrictions on the formulae allowed for association definitions:

- the only terms of sort $\mathsf{NAME}$ allowed are the identifiers used as participants in the association definition (which will be implicitly universally quantified variables),
- if a participant $p : C$ is involved in an atomic formula or term, then that atomic formula or term has to be built exclusively out of symbols defined in $C$, besides $p$.

These formulae are a logical characterisation of the interactions between components, associated to the association definitions.

*Example 3.* Let us consider the association definition given in Figure 5. Our intention is to make print servers and printers interact by:

- relating the *p-ready* read variable of a server to the *ready* attribute of its connected printer,
- relating action *dispatch*$(j)$ of a server to the action *load*$(j)$ of its connected printer.

So, the formulae we want to include in the association definition *Prints-for* are the following[5]:

$$\begin{array}{l} s.\textit{p-ready} = p.\textit{ready}, \\ \forall j \in \mathsf{string} : (s.\textit{dispatch}(j) \leftrightarrow p.\textit{load}(j)). \end{array}$$

This example shows a very basic interaction definition. It is clear though that due to the generality of the associations definition, the specifier might be able to define very complex interactions, by relating the actions and attributes of a component to the ones in another component, via arbitrarily complex formulae. Moreover, the definition of associations allows not only for binary associations, but for associations of any arity, including unary ones.

Just to illustrate how more complex interactions can be defined, suppose we decide to eliminate the redundant *ready* attribute of printers (recall that *ready* is true if and only if *job* is the empty string). Print servers still need to know somehow when the "environment" is ready for them to dispatch. We can then replace the first of the above formulae characterising the interactions associated to *Prints-for* by:

$$(s.\textit{p-ready} = \mathsf{T}) \leftrightarrow (p.\textit{job} = []).$$

[5] Note that, contrary to the way we described interactions in [3] by means of connections, we choose now to directly use formulae to characterise interactions. This allows for more generality for describing component communication.

### 4.5 Building Theories for Class Definitions

The description of classes is done based on the language defined by an underlying abstract datatype specification $\mathcal{ADT}$. Together with its signature description, a class specification has a number of explicit axioms, i.e., properties characterising the behaviour of the instances of the class.

In order to reason about class properties, we not only use the axioms of the class. The definition of the datatypes, i.e., the axioms of $\mathcal{ADT}$, play a role in class properties. So, it is natural to consider the $\mathcal{ADT}$ axioms as part of the description of a class. Note that the inclusion of the axioms of $\mathcal{ADT}$ as implicit axioms of a class specification is perfectly valid, since the alphabet associated to a class definition is a proper extension of the alphabet of $\mathcal{ADT}$.

The monotonicity of the logic guarantees that we can reason *locally* about datatype properties in $\mathcal{ADT}$, and then "import" the properties in any reasoning within any class specification.

Besides the abstract datatype axioms, there can be other implicit axioms, characterising general assumptions or properties of the specific application domains (e.g., the locality axiom [9] of components can be regarded as a general assumption about components).

### 4.6 Building Theories for Subsystems

As for basic components, in a subsystem definition we have a number of explicit axioms, which represent properties of the subsystem. However, one should not expect to be able to reason about the subsystem's properties only in terms of its explicit axioms, since if that was the case, the specifications of datatypes and basic components would be irrelevant. So, as one might expect, we need to put together the explicit axioms of a subsystem definition with the descriptions of basic components, and datatypes. For datatypes, it is easy: we simply incorporate in the description of our subsystem the axioms for the abstract datatype specification $\mathcal{ADT}_{\mathsf{NAME}}$, which itself includes the $\mathcal{ADT}$ specification. So, again, we can locally reason about datatypes, either in $\mathcal{ADT}$ or $\mathcal{ADT}_{\mathsf{NAME}}$, and then import the properties of datatypes into a subsystem *Sub*, and use them to prove other properties within *Sub*.

The slightly more difficult part is the combination of the subsystem description with the basic component descriptions. We have modified the class signatures in order to incorporate the language of components to the language of subsystems, by adding to some of their syntactic elements an extra parameter of sort $\mathsf{NAME}$. This extra parameter allows us to *relativise* the class definitions, so we can use them to represent a (varying) number of "instances" of classes in (dynamic) subsystems. Fortunately, our definition of alphabet morphisms admits this kind of "relativisation". Therefore, there exists an alphabet morphism between the alphabet of any class definition and the alphabet of an including subsystem. We now need to incorporate the class descriptions in the description of an including subsystem. The definition of the *grammar translation* in Definition 8 suggests a way of doing so. It consists in quantifying universally over a variable filling the "empty" places introduced for the extra parameters of predicate and function symbols. So, for instance, consider the axiom:

$$\Box[\forall j \in \mathsf{string} : \mathit{Printer} \wedge \mathit{load}(j) \rightarrow \mathit{ready} = \mathsf{T}]$$

which we have in the printers specification. The formula resulting from the grammar translation of the above formula, to be introduced in the including subsystem *Multiple_Printers*, is the following:

$$\forall x \in \mathsf{NAME} : \Box[\forall j \in \mathsf{string} : \mathit{Printer}(x) \wedge x.\mathit{load}(j) \rightarrow x.\mathit{ready} = \mathsf{T}].$$

Note that this translation is what we are looking for. It intuitively fits our needs. Basically, it transforms a property $\alpha$ given in a component definition $C$ into the property "all live instances of $C$ have property $\alpha$", for inclusion in a subsystem aggregating $C$.

Now we exploit the fact that the logic constitutes a $\pi$-institution. Considering the (explicit and implicit) axioms of a class definition $C$, after relativisation, as part of the set of implicit axioms associated to a subsystem definition *Sub*, allows us to reason *locally* about properties of $C$, and then import the resulting properties into *Sub*, via relativisation. The structurality property guarantees that consequence is preserved. So, if $\alpha$ is a consequence of the axioms of class $C$, then the relativisation of $\alpha$ (which roughly looks like $\forall x : \alpha$) is a consequence of the relativisation of the axioms of $C$, which are implicitly included as part of a subsystem definition *Sub* aggregating $C$.

Figure 7 shows the relationships among the different theories describing our sample subsystem. Arrows denote the existence of an alphabet morphism between the corresponding alphabets, and preservation of consequence in the source theory by the target. Arrows labelled with *id* indicate that no change is performed in the language of the source for inclusion in the target. Arrows labelled with *rel* indicate that a relativisation is necessary.

$$\begin{tikzcd}
 & \mathit{Multiple\_Printers} & \\
\mathit{Printer} \arrow[ur, "rel"] & \mathcal{ADT}_{\mathsf{NAME}} \arrow[u, "id"] & \mathit{Server} \arrow[ul, "rel"'] \\
 & \mathcal{ADT} \arrow[ul, "id"] \arrow[u, "id"] \arrow[ur, "id"'] &
\end{tikzcd}$$

**Fig. 7.** Relationships among theories of a system description.

It is important to note that class and subsystem definitions are conceptually very similar. In particular, both contain a signature, which defines an alphabet, and (implicit and explicit) formulae, which generate a theory (i.e., their semantics are defined in the same way). We could think then of using subsystem definitions as components of higher level subsystem definitions, leading to a *hierarchical* organisation of a system in terms of subsystems and basic components. A higher level subsystem would relativise the languages of other (more basic) subsystem definitions, in order to logically represent a notion of subsystem instances[6]. Associations relating instances of subsystems could be used to achieve subsystems interaction.

Being able to hierarchically organise systems in terms of subsystems and basic components would allow us to use subsystem definition as a new coarser grained unit of modularisation. This, in the context of logical specifications of systems, is a major benefit, since it allows us to exploit (hierarchical) subsystems as a way of further localising reasoning to the relevant parts of a system. We plan to explore the definition and use of hierarchical subsystems in depth.

*Formulae Related to Association Definitions.* Association definitions have some associated formulae, with the participants of the associations as free variables. In order to incorporate the formulae associated to an association $R$ in an including subsystem *Sub*, we simply quantify the free variables of the formulae universally, and force them to be related via the flexible predicate $R$. Thus, if $\alpha(x, y)$ is a formula characterising interactions of association $R$, where $x$ and $y$ are the free variables of $\alpha(x, y)$ representing the participants, then the formula:

$$\Box[\forall x, y : R(x, y) \to \alpha(x, y)]$$

is included in *Sub*, clearly characterising the fact that components referenced by $x$ and $y$ *must* collaborate according to the connections of $R$ if they are "connected" by $R$.

So, for our sample *Prints-for*, the formulae that will be implicitly included in *Multiple_Printers* are the following:

$$\Box[\forall x, y : \textit{Prints-for}(x, y) \to (x.\textit{p-ready} = y.\textit{ready})],$$
$$\Box[\forall x, y : \textit{Prints-for}(x, y) \to (\forall j \in \mathsf{string} : (x.\textit{dispatch}(j) \leftrightarrow y.\textit{load}(j)))].$$

As for the theories associated to class definitions, there might be a number of other implicit axioms to include in a subsystem, related to general assumptions or properties of the application domain. For instance, the locality of the subsystem [3] is generally assumed to be a property of subsystems, and therefore might be implicitly included in the theory describing any subsystem. Axioms typing the

[6] The careful reader might have noticed that we only used one extra argument in relativising basic component alphabets as we included them in subsystems. However, when this process is iterated, including a subsystem instance in a higher level subsystem, we will have need of further arguments for relativisation, allowing several "dots" in the dot notation.

associations can be also considered as general assumptions (e.g., if $x$ and $y$ are related via *Prints-for*, then $x$ must be the name of a live server and $y$ the name of a live printer).

## 5 Reasoning about Specifications

In the previous section, we described how the logic can be employed to specify different aspects of a dynamically reconfigurable component based system. Moreover, we showed how specifications can be hierarchically modularised, in terms of datatypes, components and subsystems. We now describe how reasoning can be carried out, showing in particular how the tool support for the Manna-Pnueli logic (which the presented logic is a variant of) can be used to *partially* reason about this kind of specification.

The difference between our logic and the Manna-Pnueli logic is that we allow for more general flexible symbols. While *local variables* (a special kind of 0-ary function symbol) are the only symbols interpreted in a state dependent way in the original logic, our variant allows for general function and predicate symbols to be flexible. However, our modifications can be somehow "implemented" in the original logic. As we briefly described in a previous section, the more complex flexible symbols which we have introduced can be represented by defining special datatypes.

Our first layer of specification is the datatypes layer. Since datatypes do not use flexible symbols, they can be specified directly in the original logic. So, let us assume that we count on a characterisation of $\mathcal{ADT}$ in the Manna-Pnueli logic[7].

The next layer is the one for class definitions, in which we describe the behaviour of basic components. In this layer, we make use of the first kind of more general flexible symbols, namely flexible predicates. Flexible predicates are used to characterise actions of components. They can be "implemented" in the Manna-Pnueli logic, as follows:

– Make sure that the basic datatypes specification includes a specification of booleans (this is obviously the case in the STeP tool).
– For each action symbol $a(S_1, \ldots, S_k)$, introduce a new datatype *TypeOfa*, which characterises functions from $S_1 \times \ldots \times S_k$ to booleans.
– Action $a$ is then defined to be a *local variable* of type *TypeOfa*.

The characterisation of functions is simple. Functions can be specified in a similar way to that used to specify arrays, i.e., with function/array "update" and "read" operations. The STeP specification in Figure 8 is an example of how this kind of datatype can be defined. It is the definition of `ITB`, a datatype which can be used to characterise actions with one integer parameter. Note that technically this specification does not fit in our formalism, since it uses simplification rules as well as logical axioms.

[7] Note that the STeP tool incorporates sophisticated mechanisms for reasoning about basic datatypes. Since these are safely combined with the Manna-Pnueli logic, they could also be used to specify the datatypes in $\mathcal{ADT}$.

```
SPEC
type ITB == empty | write :: pos: int, val: bool, fun: ITB

value read : int * ITB --> bool

variable x,y : int
variable z : bool
variable g : ITB

AXIOM : [] Forall i : int . read(i,empty) = false

AXIOM : [] Forall i,j : int . Forall f : ITB .
           Forall b1,b2 : bool .
                   write(i,b1,write(j,b2,f))=
                         if i = j then write(i,b1,f)
                                  else write(j,b2,write(i,b1,f))

REWRITE r1 : read(x,empty) ---> false

SIMPLIFY : read(x,write(y,z,g)) ---> read(x,g) if !(x = y)

SIMPLIFY : read(x,write(y,z,g)) ---> z if (x = y)

REWRITE r4 : write(x,false,empty) ---> empty
```

**Fig. 8.** An example of a datatype definition for action characterisation.

The modularisation mechanisms of STeP can be exploited to organise component specifications. So, for instance, one might specify datatypes in a theory, and then import that theory in others containing specifications of components.

The last layer of specification we described is the one for subsystems. In this layer we use both flexible predicates and flexible functions. There is an extra datatype, NAME, that is available for subsystems. It is not a complication to define this extra datatype. Flexible predicate symbols are also used to denote actions (both the actions originating in component definitions and the reconfiguration actions inherent to the subsystem). They can be characterised in the same way as actions of basic components.

Complex flexible function symbols are used to characterise the attributes and read variables of the basic components, embedded in an including subsystem (recall that they need to be relativised to names of instances). The procedure for implementing a flexible function symbol $f : S_1 \times \ldots \times S_k \to S$ is similar to the one applied for characterising flexible predicates: a type *TypeOf_f* is specified as a basic type (characterising functions from $S_1 \times \ldots \times S_k$ to $S$), and $f$ is declared as a local variable of type *TypeOf_f*.

The theorem prover of the STeP tool can be very helpful for proving properties of systems specified using this formalism. However, there is a major shortcoming that cannot be overcome with the present status of the tool. As we explained in previous sections, the formalism was devised with special emphasis on *localising* reasoning to the relevant parts of a specification. Thus, certain reasoning regarding a component's properties might be carried out *within* that component's theory, and then the properties can be promoted to an including subsystem, via a relativisation translation[8]. The STeP tool cannot deal with relativisation translations, and therefore it would be necessary to *redo* the proofs of component properties within subsystems.

This problem suggests studying a proof-theoretic counterpart of our structurality theorem. Having this result, and given a proof of a property $\alpha$ from the axioms of a component, we would count on a way of systematically constructing a proof for the relativisation of $\alpha$ from the relativisation of the axioms of that component. Consider the fragment of a sketch of a proof, in a natural deduction style, of a property in the *Multiple_Printers* subsystem, shown in Figure 9. This shows, in step 24, how we can reason in our logic, by importing properties of component definitions via relativisation (although our justification for this is, at the moment, semantic). Note that step 27 imports a property of *Prints-for*, but without the need for relativisation. A proof-theoretic structurality would allow us to prove properties in this manner, but a property importation mechanism (if proof is carried out in a top-down fashion, it would be a proof decomposition mechanism) representing structurality would have to be incorporated into the STeP tool.

We are confident that proof-theoretic structurality holds, for the present proof calculus of the Manna-Pnueli logic, but this remains to be proved. Even in the case the property does not hold, we could modify the proof calculus of the logic with such a rule, whose soundness is guaranteed by our semantic structurality theorem.

## 6 Conclusion

We have presented in detail an adaptation of a first-order temporal logic, and justified its suitability for the specification of component based systems with support for run time reconfiguration. This justification was done by showing that the logic constitutes a $\pi$-institution, in which language translations admit relativisations. We showed how different aspects of a component based system can be characterised by formulae in this logic, and how specifications can be modularised in terms of datatypes, components and subsystems. Moreover, we discussed how the structurality property of the logic can be exploited to localise reasoning to the relevant parts of a system's specification. We also discussed

[8] Note that it is not necessarily the case that, while reasoning from a subsystem's theory, we can always restrict reasoning regarding a property relevant only to a certain component to that component's theory. It is known that components can have *emergent properties* [8] (properties that they do not have if considered in isolation) when forming part of a system, and that such properties can only be derived from the (sub)system specification.

| | |
|---|---|
| 20- $\ldots$ | |
| 21- $Server(s)$ | Subsystem Prop. |
| 22- $Prints\text{-}for(s, p)$ | Hyp. |
| 23- $s.send()$ | Hyp. |
| 24- $\forall x : \Box[Server(x) \wedge x.send() \rightarrow x.dispatch(\mathrm{hd}(x.p\text{-}queue))]$ | *Server* Prop. |
| 25- $Server(s) \wedge s.send() \rightarrow s.dispatch(\mathrm{hd}(s.p\text{-}queue))$ | $\Box$ Elim. & Inst. 24 |
| 26- $s.dispatch(\mathrm{hd}(s.p\text{-}queue))$ | MP 21,23;25 |
| 27- $\forall x, y : \Box[Prints\text{-}for(x, y) \rightarrow$ $(\forall j \in \mathsf{string} : x.dispatch(j) \leftrightarrow y.load(j))]$ | *Prints-for* Prop. |
| 28- $Prints\text{-}for(s, p) \rightarrow (\forall j \in \mathsf{string} : s.dispatch(j) \leftrightarrow p.load(j))$ | Inst. & $\Box$ Elim. 27 |
| 29- $(\forall j \in \mathsf{string} : s.dispatch(j) \leftrightarrow p.load(j))$ | Inst. & MP 22,28 |
| 30- $s.dispatch(\mathrm{hd}(s.p\text{-}queue)) \leftrightarrow p.load(\mathrm{hd}(s.p\text{-}queue))$ | Inst. 29 |
| 31- $p.load(\mathrm{hd}(s.p\text{-}queue))$ | MP 26,30 |
| 22- $\ldots$ | |

**Fig. 9.** Fragment of a proof illustrating the importing of a component's properties.

some required extensions to the current tool support for the Manna-Pnueli logic (which the presented logic is a variant of) to achieve localisation of reasoning.

Two main topics are among the priorities to continue our research. One is related to the development of a proof system for the presented logic, in which, as we argued, a proof-theoretic structurality property has to be available. The other is of a more methodological nature. It has to do with exploring suitable ways of hierarchically defining subsystems in terms of more basic subsystems and components. This is especially important, since in the context of logical specifications (and with the availability of a suitable proof system), it would allow us to exploit subsystems as a way of further localising proofs to the relevant parts of a specification.

The generality of our characterisation of interaction suggests another line of research, exploiting this generality to represent more complex notions of associations, with applications in, for instance, software architectures and aspect oriented approaches.

## References

1. M. Abadi and Z. Manna, *Nonclausal Deduction in First-Order Temporal Logic*, Journal of the ACM 37(2): 279-317, 1990.
2. N. Aguirre and T. Maibaum, *A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems*, in Proceedings of the 17th International Conference Automated Software Engineering ASE 2002, Edinburgh, United Kingdom, IEEE Press, 2002.
3. N. Aguirre and T. Maibaum, *A Logical Basis for the Specification of Reconfigurable Component-Based Systems*, in Proceedings of Fundamental Approaches to Software Engineering FASE 2003, Warsaw, Poland, LNCS 2621, Springer, 2003.
4. R. Allen and D. Garlan, *Formalizing Architectural Connection*, in Proceedings ICSE '94, Sorrento, Italy, 1994.

5. M. Barr and C. Wells, *Category Theory for Computing Science*, Third Edition, Les Publications CRM, 1999.
6. 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.
7. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of EATCS Monographs on Theoretical Computer Science, Springer, Berlin, 1985.
8. J. Fiadeiro, *On the Emergence of Properties in Component-Based Systems*, in Proceedings of AMAST'96, M.Wirsing and M.Nivat (eds), LNCS 1101, Springer-Verlag, 1996.
9. 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.
10. 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-Verlag, 1988.
11. D. Garlan, *Software Architecture: A Roadmap*, in The Future of Software Engineering, A. Filkenstein (ed), ACM Press, 2000.
12. D Garlan, R. Monroe and D Wile, *ACME: An Architecture Description Interchange Language*, in Proc. of CASCON'97, 1997.
13. J. Goguen and R. Burstall, *Institutions: Abstract Model Theory for Specification and Programming*, Journal of the ACM 39(1): 95-146, ACM Press, 1992.
14. P. Inverardi and A. Wolf, *Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine*, IEEE Transactions in Software Engineering, 1995.
15. Z. Manna and A. Pnueli, *Verification of Concurrent Programs: A Temporal Proof System*, Technical Report No. STAN-G-83-967, Department of Computer Science, Stanford University, 1983.
16. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.
17. Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems - Safety -*, Springer, 1995.
18. N. Medvidovic, *ADLs and Dynamic Architecture Changes*, in Proceedings of the Second Int. Software Architecture Workshop (ISAW-2), 1996.
19. N. Medvidovic and R. Taylor, *A Framework for Classifying and Comparing Architecture Description Languages*, In ESEC-FSE'97, 1997.
20. B. Meyer, *Object-Oriented Software Construction*, Second Edition, Prentice-Hall International, 2000.
21. J. Ostroff, *Temporal Logic for Real-Time Systems*, Advanced Software Development Series, Research Studies Press Ltd, John Wiley and Sons, 1989.
22. D. Parnas, *A Technique for Software Module Specification with Examples*, in Communications of the ACM, 15(5), 1972.
23. D. Parnas, *On the Criteria to be Used in Decomposing Systems into Modules*, in Communications of the ACM, 15(12), 1972.
24. M. Wermelinger, A. Lopes and J. Fiadeiro, *A Graph Based Architectural (Re)configuration Language*, in ESEC/FSE'01, V.Gruhn (ed), ACM Press, 2001.