Chapter 1

Temporal Specifications of Component Based Systems with Polymorphic Dynamic Reconfiguration

Nazareno Aguirre[†] and Tom Maibaum[‡]

[†] *Departamento de Computación, FCEFQyN,*
Universidad Nacional de Río Cuarto,
Enlace Rutas 8 y 36 Km. 601, Río Cuarto (5800), Córdoba, Argentina
naguirre@dc.exa.unrc.edu.ar

[‡] *Department of Computing & Software, McMaster University,*
1280 Main St. West, Hamilton L8S 4K1,
Ontario, Canada
tom@maibaum.org

In this chapter, we present a formal characterisation of component based systems with support for polymorphic dynamic reconfiguration. By *dynamic* reconfiguration we mean, as usual, changes in the system architecture at run time. By *polymorphic* reconfiguration we mean that reconfiguration operations may concern different types of components or connections, exploiting an inheritance relationship over components, as in object orientation.

The formal characterisation of component based systems is based on a first-order temporal logic. The logic is a variant of the Manna-Pnueli logic, expressive enough for straightforward specification of component types, connector types and dynamic amalgamations of components. On top of this logic, and in the form of a (rather low level) specification language, we build the necessary machinery for specifying components, connectors and amalgamations, together with inheritance and polymorphism.

1.1. 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 were eventually integrated into programming methodologies [24, 25]. The advantages that structuring software systems into modules have in all phases of software development, from analysis to maintenance, were instantly recognised and have strengthened over the intervening decades.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

In the past decade or so, a new branch of software engineering emerged with the name software architectures (SAs) [4, 15]. This branch reemphasises the notion of module, or component, at a perhaps higher level of abstraction than that normally used in other modern modelling (and programming) methodologies, such as object orientation. Software architectures suggest the modelling of systems structure in terms of components related by means of *connectors*. Software architectures thus introduce a second modularisation concept to accompany that of components, the connector. The motivating principles are the very same ones originally motivating modularisation techniques. Software architectures notably differ from object orientation in the way interaction is represented. In software architectures, interaction is typically defined externally to components, which has the advantage of explicitly showing the structural appearance of systems. In object orientation, on the other hand, interaction can be obscured because it can be implemented via "feature calling" (of other parties in the interaction) within the interacting classes [6]. This is a problem of object oriented programming and modelling languages that several researchers have acknowledged. Rumbaugh's relational object oriented language [26], Andrade and Fiadeiro's coordination contracts [5], and various design patterns [14], are examples addressing this deficiency of object orientation.

The increasing focus on higher level structural descriptions of systems led to the development of a special type of specification language, called *architecture description language* (ADL) [23], The purpose of ADLs is to describe the architectural aspects of software systems, so properties of the specified systems, especially those involving architectural information, can be analysed.

Modern applications typically require a feature that some ADLs are able to deal with, namely *dynamic reconfiguration*. Dynamic reconfiguration refers to the run time modification of the system's structure [22]. Although this is not an inherent feature of software architectures, it appears frequently and naturally in the design of systems, perhaps due to the success of object oriented methodologies and programming languages, where it certainly is intrinsic.

While ADLs provide constructs for modelling the architecture of a system, and some of them also allow for the description of possible changes to it (usually via operations that may modify the system's structure at run time), they often do not directly support (within the language) reasoning about possible system evolution. More precisely, some ADLs support the definition of components and interconnections to build architectures, 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 [16, 18, 27].

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 using ADLs. In addition, the abstraction gained by using a declarative framework might allow us to study possibly more sophisticated, abstract ways of describing software architectures. We therefore proposed a temporal logic as a formal basis for the specification of component based

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

systems with support for run time reconfiguration [2]. We aim at facilitating the reasoning that is usually necessary regarding this kind of system. Besides providing direct support for reasoning, temporal logic gives us a declarative and well known language in which behavioural properties can be expressed, and which is currently used in several branches of software engineering.

We employ for this purpose a variant of a logic that Manna and Pnueli [21] proposed for specifying reactive systems. This variant, proposed by M. Abadi [1], allows for more general flexible, i.e., state dependent, symbols, and therefore is better suited for the specification of dynamically reconfigurable systems. In particular, the logic admits a derived proof rule, which enables us, if specifications are organised appropriately, to import properties from components when building (dynamic) amalgamations. This allows us to exploit the modular structure of specifications to localise reasoning to the relevant parts of these specifications, when proving certain properties.

A prototypical language based on the proposed logic is defined, where systems specifications are hierarchically organised around the following notions:

- the notion of datatypes, which constitute the lowest level in specifications,
- the notion of components, which are represented by classes that define templates for these components;
- the notion of connector types, 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 which components, which associations and which reconfiguration operations are used to define the module.

It is not our aim to propose (yet) another architecture description language, but to study an alternative declarative and formal semantics for software architectures, with direct support for reasoning. We prefer to illustrate the capabilites and expressive power of the formalism by defining a simple front-end to the 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, this language allows us to study the enhancements that our alternative semantics might provide to the specification of software architectures. In particular, we show that more powerful modelling constructs, such as class inheritance, as in object orientation, can be provided using our alternative semantics. This not only provides the well known advantages in terms of the reuse of component definitions, but also enables us to define polymorphic reconfiguration operations.

1.2. A Model of Reconfigurable Component Based Systems

We now define in some detail the model of reconfigurable component based systems that we assume. We try to preserve some of the good features of some ADLs,

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

such as declarativeness (as in Acme [17]), hierarchical composition of systems (as in Darwin [19]) and designs at a high level of abstraction (as in CommUnity [28]).

1.2.1. *Basic Components*

As described in [17], components are often meant to represent the primary computational and data storage units of systems in software architectures. This view of components is shared by Acme [17], CommUnity [28], Darwin [19], and Wright [3], for instance. In our view of reconfigurable component based systems, the smallest computational and data storage units are recognised as special, and are called *basic components*. We prefer to use the term 'basic component' because we also want to be able to build complex components out of simpler ones, as in Darwin or Acme. These complex components could then form part of even more complex components, or systems. As justified in [17, 19], being able to hierarchically structure components is an important feature, which can help in dealing with large systems through several layers of decomposition.

We consider that a component is *basic* if it is not composed of simpler components. For basic components, we take the (more abstract) model of components from CommUnity [28], and define basic components as consisting of variables, as in imperative programming languages, which define the components' internal state. We also assume that there is a set of datatypes provided, which are used as types for the variables within basic components. It is important to note that, since basic components are not defined in terms of other components, their variables must be typed with basic types, not types which represent components. (This underpins our view that, in order to achieve the desired low coupling between components in a system, any interaction between components must be defined completely externally to the components involved.)

As is the case with CommUnity's *channels* [28], a component can have input, output or local variables. These are represented as follows: components have attributes, which correspond to local or output variables, and read variables, which correspond to input variables. Local variables are differentiated from nonlocal ones by means of a component interface definition.

Basic components also encapsulate *actions*, which represent their associated computational behaviour. These actions allow a basic component to change its state, by modifying the values stored in its variables, and provide, in this way, some functionality to the system. Actions of components are assumed to be *instantaneous*, and they might have parameters. Actions which take some period of time to be completed can be modelled with 'start' and 'end' instantaneous actions, as in [12] and the specification stage of [9].

As an example, suppose we want to model a network of units which can interchange messages. These units might represent basic components, whose state is composed of a private address, attributes for storing outgoing messages, and read variables for obtaining incoming messages. Their associated behaviour might be represented by actions for producing and sending outgoing messages, and obtaining and consuming incoming ones.

The visibility of read variables, attributes and actions should be reflected when

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

components are connected. For instance, a public attribute of a component A could be connected to a private read variable of a component B, meaning that B can read A's attribute, but B's clients cannot read A's attribute through B. Clearly, it should not be allowed to connect private features (read variables, attributes or actions) of a component to private features of other components.

1.2.2. *Connectors*

One of the central characteristics of software architectures is that the interaction between components is characterised by means of *connectors* [17], *externally* to the definition of components. The purpose of connectors is to allow for the communication between different components in the system, so they can perform activities in combination. The externalisation of the interaction between components in connectors makes the structural appearance of systems (in terms of interrelated components) immediately apparent; this makes it easier to describe operations, constraints, properties, etc, concerning the structure of systems. The externalisation of component interactions and its benefits are successfully exploited in object orientation by techniques such as those based on certain design patterns [14], coordination contracts [5] and middleware technology [8].

In order to make components (and, as we will see, this will include complex components) interact, we use the notion of *coordination*, as in CommUnity. Basically, we use connectors to specify how the states and behaviours of interacting components are combined. Special cases of this kind of communication are sharing of variables and action synchronisation, but more sophisticated ways of communication are also possible. When two or more components are connected, their behaviours become *related*, in the way specified by the corresponding connector. For instance, the connector might specify that the occurrence of an action in one of the participating components enforces the occurrence of an action in another participant, but not vice versa (i.e., as in a remote procedure call).

An example of a connector might be a link between the previously described units. A link might enforce the occurrence of the retrieve operation on one of the connected units whenever the send operation occurs on the other one.

We are particularly inspired by the style of specification and component coordination used in [13] and related work, which has also been inherited by modern versions of the CommUnity design language [27].

1.2.3. *Complex Components*

In SAs, indexSA *systems* are typically seen as configurations of components related by means of connectors. We use the term *subsystem* to refer to such configurations. We do so because (dynamic) configurations of interrelated components might be themselves the component parts of bigger systems. A system can then be seen simply as a top level subsystem.

Subsystems represent complex types of components. They are complex in the sense that their internal definitions may involve simpler components (other subsystems or basic components). Subsystems can encapsulate data in the form of

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

variables, but as opposed to the case of basic components, subsystems can also build their internal state by using interacting instances of simpler components.

Subsystems also encapsulate behaviour. Subsystem actions represent the computational behaviour of a subsystem and, besides modifying the values stored in the subsystem variables, they can modify the internal structure of the subsystem, by creating or deleting instances of simpler components, or modifying the way in which these simpler components interact (e.g., creating or deleting instances of connectors).

As an example of a subsystem, we can consider a subnet, i.e., a collection of units connected in a star topology to a gateway (a different kind of basic component, which forwards messages to other subnets). Some basic attributes of a subnet could be the maximum number of units allowed in the subnet, or the number of messages sent since the subnet was started. A subnet might have (reconfiguration) operations for adding new units, or deleting existing ones.

As we said, complex components might also be built out of other complex components. As opposed to the case of Darwin [19], we do not allow for *recursion* in the definition of components, and allow only for *hierarchical* organisations of (sub)systems in terms of other subsystems or basic components. Connectors could then be defined to relate subsystems in complex configurations.

1.3. A Temporal Specification Language

Before describing the constituents of our prototypical specification language, let us describe in more detail the logic our work is based on.

1.3.1. *The Logic*

The main characteristics of the logic that underlies our prototypical specification language are:

- the logic is first-order, with sorts and equality,
- time is considered to be linear, with a discrete set of instants and an initial instant (i.e., the model of time is $\mathbb{N}$),
- besides the usual connectives and quantifiers, the logic also features the temporal operators $\bigcirc$, $\Box$, $\Diamond$ and $\mathcal{U}$,
- some function and predicate symbols (called *flexible*) are interpreted in a state dependent way, although functions and predicates with state independent interpretations (called *rigid*) are also available.

This logic is a many sorted version of a predecessor of the Manna-Pnueli logic [21], presented by M. Abadi [1]. With respect to the Manna-Pnueli logic, it is more general since it allows for flexible predicates (not available in Manna-Pnueli's) and flexible function symbols of arbitrary arity (in the Manna-Pnueli logic, only function symbols of arity zero are allowed to be flexible). These more general flexible symbols allow us, as we will show, to represent operations and attributes of components in a straightforward way.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

As shown in [1], the logic admits a sound (but not strongly sound) proof system. However, no complete proof calculus can be constructed for the logic [1]. Nevertheless, completeness can be achieved if one considers restricted notions of completeness. An example of this fact is the completeness of the proof calculus of [20] with respect to time sequences corresponding to executions of a concurrent program.

We represent basic components and subsystems as theory presentations. The proof calculus of the logic can then be used in order to reason about the consequences of the axioms of a component or subsystem specification.

1.3.2. *Organisation of the Language*

The prototypical language we present allows us to specify reconfigurable component based systems, by defining:

- basic datatypes, which serve as the types for variables of basic and complex components,
- basic component types, which, in contrast with complex component types, represent instances which are not composed of other (simpler components),
- connector types, which define the possible ways in which components might interact,
- complex component types, which are called subsystems; these represent the instances of components whose internal state is defined in terms of a dynamic set of interacting simpler components.

Specifications are modularly organised in layers, from datatype specifications to the specification of architectural subsystems. Subsystems might be composed of other simpler subsystems, in a non recursive way (in contrast with practice in Darwin, for instance). Thus, we permit a hierarchical approach in the organisation of reconfigurable component based systems. As we will show, we can use a derived inference rule of the logic in order to relate deduction in different layers of a specification.

1.3.3. *A Specification Problem*

In order to introduce the language and its semantics, we will use the previously described example of communicating units. Let us describe it in more detail. Suppose we want to model and analyse a network of *units*, which can interchange messages. Each unit has an associated address, which is supposed to be unique to a unit in the system. Messages are sent by units to other units. The address of the destination unit is included within each message. Also, there is a special message, called *null*, whose associated destination address is undefined.

Units are organised into *subnets*. Subnets contain a non-empty set of units, organised in a star topology. The unit at the centre of the star is of a special kind, and is called a *gateway*. Gateways receive and send messages from and to other units outside the subnet. All other units in a subnet communicate through the gateway. Gateways are units as well, so they can receive and send messages of their

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

own. New units can be added to a subnet, and existing units can be deleted, except for the gateway.

A gateway recognises whether a message is addressed to a unit within its subnet by checking its *netmask*. The netmask allows a gateway to decide whether a given address *belongs* to the set of valid addresses for its subnet or not. Note that it is not necessarily true that the whole netmask "address space" has to be covered; in other words, there might be valid addresses that actually correspond to the netmask of a subnet, but no live unit with that address resides in the subnet. A further requirement is that no address must correspond to more than one netmask in the subnets of the system.

The whole system is basically a collection of interconnected subnets. Subnets are also connected in a star topology, with a gateway in the centre (that we call a *router*). It would be important to allow the units in the system to send messages to other units possibly outside the system, although we will not use such a feature here. An interesting capability one might want to provide for the system is the possibility of dynamically adding new subnets, or possibly detaching existing subnets. An informal diagrammatic view of a system is shown in Fig. 1.1. Normal units are labelled with indexed `u`'s, gateways with indexed `g`'s and the only router with `r1`. The region within the dotted circle corresponds to a subnet; so, all units within the subnet (including the gateway) belong to the address space determined by the gateway's netmask.

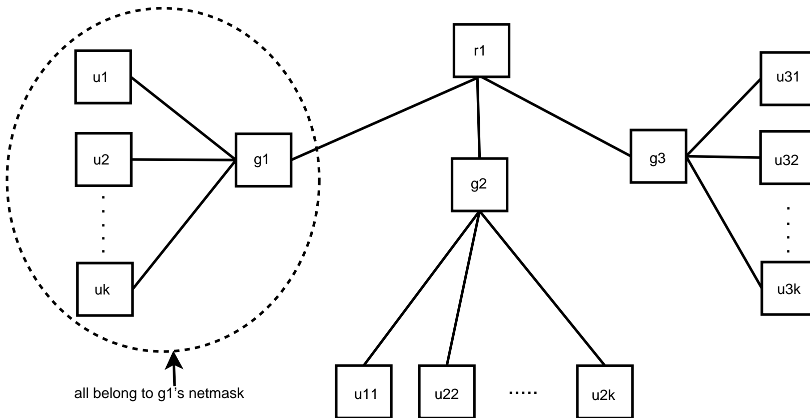

Fig. 1.1. A graphical representation of a network of units

1.3.4. *Specification of Datatypes*

Basic components build their state by means of variables, as in imperative programming languages, whose types are defined in the abstract datatypes specification. The

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

logic can be used to build first-order characterisations of abstract datatypes, in a way similar to that used in *algebraic specifications* [10]. Flexible, i.e., state dependent, symbols are not necessary for the specification of abstract datatypes (flexible symbols are reserved for the specification of the *state* of components and configurations). Therefore, the specification of abstract datatypes consists of a *static* temporal theory presentation, i.e., a temporal theory presentation over a language without flexible symbols.

Let us assume that we count on a datatype specification $\mathcal{ADT}$, containing descriptions for the standard basic datatypes, such as integers, natural numbers, booleans, etc, with their usual operations. For the sake of modelling the above described system, we assume that datatypes $\mathsf{message}$ and $\mathsf{address}$ are provided within $\mathcal{ADT}$. We also assume that the operations $null :\rightarrow \mathsf{message}$ (which represents the 'empty' message) and $dest : \mathsf{message} \rightarrow \mathsf{address}$ (which 'extracts' the address contained in a message) are available. No special axioms are required for these datatypes.

1.3.5. *Specification of Basic Components*

Basic components are one of the basic building blocks of software architectures. We intend to use temporal logic theories to describe components, as in [11]. In dynamically reconfigurable systems, a varying number of "instances" of the same type of component can be involved. Then, we want a way of describing templates of these components, rather than the components themselves (this is, in fact, the approach in most ADLs). We call these descriptions *class definitions.* Class definitions are modularly built on top of an underlying datatype specification, $\mathcal{ADT}$ in our case.

Possible counterparts of class definitions in some architecture description languages are (basic) *component types* in Darwin [19]. *component definitions* in Acme [16] and in Wright [3] (within styles) , and *component designs* in CommUnity [27].

A class specification (type for basic components) is composed of:

- a set of typed read variables (entry points for the components, used for communication),
- a set of typed attributes, which represent the variables that constitute the *state* of the instances of the class,
- a set of parameterised actions, which represent the behaviour associated to the instances of the class.

The intended behaviour of the instances of a class specification is defined by means of temporal formulae. These formulae will, in particular, indicate what is the effect of actions in the state of the instance, i.e., in the values of the attributes. Given a class C, a temporal axiom for it is a formula over the alphabet obtained by combining the alphabet of the datatype specification $\mathcal{ADT}$ with:

- the read variables and attributes of the signature treated as flexible 0-ary function symbols of the corresponding sort; the flexibility of these symbols represents the possible change of the values stored in read variables and attributes.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

- The actions of the signature treated as flexible predicate symbols; the truth of flexible predicate $a(\overline{x})$ at a given instant i represents the occurrence of action a at i, with the arguments $\overline{x}$.
- The class name C treated as a 0-ary flexible predicate. The truth of C at a given instant i represents the "activeness" of the corresponding component at that instant.

The flexible predicate C, named after the name of the class, is used to represent the *activeness* of the corresponding "object" or instance. Note that C represents a kind of structural information about 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 one usually requires an instance to have a property *only* while it is active. Moreover, the use of the activeness predicate is central to our approach to the specification of reconfigurable systems. In order to understand why, consider a simple property P of a component type C. This property should be characterised within C's theory by an axiom of the form:

$$C \to P$$

As will be shown later on, in order to build theories representing aggregations of instances of C, this formula is *relativised*, into the form:

$$\forall x : C(x) \to P(x)$$

which adequately indicates that, for every x, if x denotes the name of a "live" instance of class C, then x has the property P.

A class definition might also include an *interface*, which is simply a list of those read variables, attributes and actions that are visible from the outside of the class.

The most trivial basic component that we can recognise in our problem is *unit*. As we described units, they have the capability of sending and receiving messages to and from other units. However, we do not allow a class or any component type (e.g., subsystems) to directly reference other components, not contained in it. So, the communication part of a unit has to be characterised via read variables, attributes and connector definitions.

A possible specification of units is composed of the following:

- Read variables:
  - a boolean-typed read variable, called *in*, which indicates to a unit whether there is an incoming message ready to be obtained (from the "environment"),
  - a boolean-typed read variable, called *out*, which tells a unit whether the "environment" is ready to receive an outgoing message (produced by the unit).
- Attributes:
  - an address-typed attribute, called *addr*, meant to contain the address of the unit (which, let us recall, is supposed to be unique to the unit),
  - a message-typed attribute, called *curr-out*, where outgoing messages are stored before sending them,

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

  - a message-typed attribute, called *curr-in*, where the incoming messages addressed to the unit are stored, before consuming them.

- Actions:
  - action *u-init*(address), which initialises a unit by setting its address to the one passed as a parameter, and setting attributes *curr-in* and *curr-out* to the *null* message,
  - action *prod*(message), which produces its argument message, storing it in the *curr-out* attribute, ready to be sent,
  - action *send*(message), which sends a previously produced message to the environment,
  - action *get*(message), which obtains, from the environment, an incoming message,
  - action *cons*(message), which consumes a previously obtained message, provided that it is addressed to the unit,
  - action *rem*(), which removes a previously obtained message, provided that it is *not* addressed to the unit.

The class specification *Unit* is shown in Fig. 1.2. Note that neither its read variables nor the *addr* attribute belong to the class' interface. The class' interface is defined by the **Exports** clause, and as we said, indicates the visibility of the class' attributes, read variables and actions. If the **Exports** clause is missing in a class definition, it is assumed that all the class' features are exported. Axiom 1 specifies the intended behaviour of action *u-init*(message) (note how we use the 'next' temporal operator to specify the state after the occurrence of an action). Axiom 2 indicates that this action can be called at most once per lifetime of a unit (a constraint not originally stated in the problem description), using the derived 'weak until' temporal operator. Axiom 3 is a *locality* condition, indicating that only *u-init*(message) can update the *addr* attribute.

The general flexible predicate symbols of the logic allow us to represent parameterised actions of components in a straightforward way.

Axioms 4-5, 6-7 and 8-9 specify the *get*(address), *cons*(address) and *rem*() operations, respectively. Axiom 10 is, again, a locality condition, indicating that *get*(address), *cons*(address) and *rem*() are the only operations that can update the *curr-in* attribute.

Axioms 11-12 and 13-14 specify the *send*(address) and *prod*(address) operations, respectively. Finally, Axiom 15 is another locality condition, restricting attribute *curr-out* to be modified only by actions *send*(address) and *prod*(address).

Note that all these axioms are subject to the activeness of the unit. The careful reader might also notice that we have no axiom specifying the condition that an address must be unique to a single unit. The reason for this is that this condition corresponds to a structural constraint, which is beyond the language of the *Unit* component. Therefore, it will have to be specified within the containing subsystem, *SubNet*.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

1.3.5.1. *Semantics of Classes*

A class specification C is interpreted as a theory presentation, over the alphabet composed of the $\mathcal{ADT}$ specification extended with C's read variables, attributes, actions and activeness predicate. The axioms of the presentation are obtained by putting together:

- the axioms explicitly provided for the class definition,
- the axioms $Ax_{\mathcal{ADT}}$ of the datatype specification $\mathcal{ADT}$,
- a special implicit axiom, called the *locality axiom* for the specification, whose general form is:

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

  where C is the component's name (i.e., its activeness predicate), Act is the set of exported actions, and Att is the set of attributes of the component.

Intuitively, the locality axiom for a class specification expresses the fact that in every state in which the component is actively involved, either one of the actions is triggered or all the attributes remain unchanged. This axiom was originally proposed in [11], and enforces a kind of encapsulation. Locality axioms are sometimes referred to as "frame axioms" or "frame properties". We have imposed stronger locality conditions as part of the specification of units.

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.

Another important basic component is the *gateway*. Gateways have the purpose of "forwarding" messages that arrive to a subnet to its constituent units, and send to other subnets the "internal messages" addressed to "external units", i.e., to units which reside outside the gateway's subsystem. As stated in the problem description, gateways also behave as units, i.e., they can send messages of their own, and receive messages addressed to them.

In order to specify gateways, we can take the specification of units, and extend it with further read variables, attributes and actions:

- Read variables:
  - a boolean-typed one, named *int-in*, which indicates to a gateway whether there is a message from one of the "internal units" (the units within the subnet) ready to be obtained,
  - a boolean-typed one, named *int-out*, which tells a gateway if the "internal units" are ready to receive a message from the gateway.
- Attributes:
  - an address → boolean one, named *netmask*, which characterises the set of "valid" addresses for units internal to the corresponding subnet,

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

  - a message-typed one, named *int-curr-in*, which holds a just received message from the internal units of the corresponding subnet,
  - a message-typed one, named *int-curr-out*, which holds a message ready to be sent to units internal to the corresponding subnet.

- Actions:
  - action *int-prod*(message), which produces its argument message, storing it in the *int-curr-out* attribute, ready to be sent "internally",
  - action *int-send*(message), which sends a previously "internally produced" message to the environment (actually, to other units in the subnet),
  - action *int-get*(message), which obtains from the environment (actually, from some unit internal to the corresponding subnet) an incoming message,
  - action *int-cons*(message), which consumes a message previously obtained from other units internal to the subnet, provided that it is addressed to the gateway,
  - action *int-rem*(), which removes a message previously obtained from other units within the subnet, provided that it is *not* addressed to the unit.

The behaviour of some of the actions originating in *Unit* has to be enhanced. Operation *rem*() has a new associated behaviour: if the message in *curr-in*, let us say m, is not addressed to the gateway but it *does* correspond to the netmask (i.e., it is a valid address of the subnet), then *int-prod*(m) is "called", so that the message is forwarded to the units of the subnet. Note that this does not violate any of the previous axioms regarding action *rem*().

A similar behaviour is the one associated to action *int-rem*(), this time referring to the "internal interface" of the gateway. If the message in *int-curr-in*, let us say m, which was previously obtained from other units in the subnet, is not addressed to the gateway, then there are two alternatives:

- if m corresponds to the netmask of the subnet, then *int-rem*() removes the message from *int-curr-in* and "calls" *int-prod*(m), in order to forward it to the units internal to the subnet,
- if m does not correspond to the netmask of the subnet, then *int-rem*() removes the message from *int-curr-in* and "calls" *prod*(m), in order to forward it to the units outside the subnet.

The rest of the operations behave in the same way as the operations originating in *Unit*, but managing the internal interface of the gateway.

1.3.6. *Extending Class Specifications*

A first choice for specifying gateways would be to give a separate theory presentation, independent of the specification of units. However, it is clear that gateways share many properties with units. In fact, as was stated in the problem description, gateways are special units, which provide an extra "interface". We could, therefore, try to represent gateways by making use of an *inheritance* mechanism, as in object orientation.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

In order to define an inheritance relationship between components (in this case, basic components), we have to agree on what is meant by correct extension of a component definition. In object orientation, the intended meaning associated with the extension of a class A by a class B is often associated with the following facts:

(1) instances of B provide *at least* all the "services" that instances of A provide, but instances of B might provide more "services" (i.e., extra behaviour),
(2) if instances of A are replaced by instances of B (in contexts in which instances of A are employed), then the observable behaviour should not be altered (the substitution principle).

In order to characterise these points, we say that a class C_{Sub} *extends* a class C_{Super} if and only if the following conditions are satisfied:

- the mapping τ, that maps
  - predicate symbol C_{Super} to C_{Sub},
  - any other element of the alphabet corresponding to C_{Super} to itself, as an element in the alphabet corresponding to C_{Sub} (i.e., τ works as the identity injection for all symbols of class C_{Super}, except fot the predicate C_{Super} itself),

  is an alphabet morphism between the alphabet corresponding to C_{Super} and the alphabet of C_{Sub},
- The interface of C_{Super} is respected by C_{Sub}; i.e., all symbols exported by C_{Super} are also exported by C_{Sub}.
- τ is a *theorem preserving* language translation, between the theory of the superclass C_{Super} restricted to the sublanguage of its interface plus predicate C_{Super}, and the theory of the subclass C_{Sub} restricted to the sublanguage of the interface of C_{Super} plus its activeness predicate C_{Sub}.

Note that the first of the above conditions requires somehow that the subclass *extends* the language of the superclass, except for the predicate symbol denoting activeness of the superclass instances, which is appropriately translated into the corresponding predicate in the subclass. In other words, the signature of the superclass is embedded in the subclass. It is also worth noting that the conditions for valid inheritance do not forbid either interface or signature expansion of the superclass by the subclass.

Our intention with the last of the above conditions is to logically represent what preservation of the "abstract meaning" of actions is. The restriction of meaning preservation to the interface of the superclass has to do with the fact that what is "internal" to a class (i.e., not exported) might be modified by the superclass, as long as the observable behaviour of the exported operations of the superclass in maintained. But, how do we define *observable behaviour*? Interfaces help with this. The behaviour of a component is characterised by the theorems of the component; the observable behaviour is then the set of theorems restricted to the sublanguage of the interface. We require the subclass to respect the observable behaviour of the superclass, which means that the theory of the subclass should be an extension

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

of the observable behaviour of the superclass. Hence the third condition for valid inheritance.

Using this definition of class extension, we can define the *Gateway* class as an extension of *Unit*. The corresponding class specification is shown in Fig. 1.3. We have chosen to make a simplification to gateways: their internal netmask cannot change. This restriction (characterised by Axiom 2) was made simply to keep *Gateway*'s specification simple; it is not difficult to extend gateways with further operations in order to manage their corresponding netmasks.

The "forwarding" capability of gateways is characterised by Axioms 15-17. Forwarding is associated with actions *rem*() and *int-rem*(), i.e., the actions responsible for "removing" incoming messages of the two "interfaces" of the gateway, when they are not addressed to the gateway. Note how Axioms 15-17 look at the netmask in order to decide what to do with an incoming message. We might also want to add two further axioms to the specification of gateways:

- if a message is produced in order to be sent "outside" the corresponding subnet, then its destination address does not correspond to the netmask of the gateway,
- if a message is produced in order to be sent "inside" the corresponding subnet, then its destination address must correspond to the netmask of the gateway.

These can be specified by the following formulae:

$$\Box[\forall x \in \mathsf{message} : \mathit{Gateway} \wedge \mathit{prod}(x) \rightarrow \neg \mathit{netmask}(\mathit{dest}(x))]$$
$$\Box[\forall x \in \mathsf{message} : \mathit{Gateway} \wedge \mathit{int\text{-}prod}(x) \rightarrow \mathit{netmask}(\mathit{dest}(x))]$$

We assume that all axioms (explicit and implicit) corresponding to *Unit* are inherited by *Gateway* (although we do not write them explicitly). Then, we are trivially in the presence of valid class extension in this case, since all axioms of *Unit* are preserved (since the proof calculus for the logic is monotonic).

1.3.7. *Specification of Connectors*

We have just described the way component types can be defined, by means of class definitions. We choose to define these class definitions as independent units. With the exception of inheritance, classes cannot refer to other classes within their definitions. Even in the case of inheritance, we can still see classes as completely independent units, by incorporating the implicitly inherited behaviour from the superclasses in the subclasses. This is crucial, since from a logical point of view, it allows us to reason about component properties independently of the rest of the system.

Now we want to start defining dynamic aggregations of components. But of course we need ways of making components interact. In order to allow gateways to forward messages to other units in the subnet, we need to define a kind of communication link between them. A communication link within a subnet will have:

- at one end, a gateway instance, the centre of the star,
- at the other end, a unit instance, which, in principle, could be another gateway.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

We represent communication links (i.e., connectors) by means of association definitions. Associations are composed of a set of participants (typed with class names) and connections, which are special formulae that relate the participants. For our specification problem, the specification of association *Link* is shown in Fig. 1.4. Note that the second participant, t, has been defined to be of "type" **Unit**. This type corresponds to the *coverage* of *Unit*, i.e., it characterises instances of *Unit* or subclasses of *Unit* (*Gateway*, for instance). We show how this predicate is characterised later on. In this association, it allows the second participant to be a unit or gateway. Note also that, probably against our first intuition, we have used implication instead of bi-implication in some of the connections. An example of this is the following:

$$(t.\mathit{curr\text{-}out} \neq \mathit{null}) \rightarrow (s.\mathit{int\text{-}in} = \mathsf{T})$$

which basically indicates that if the *curr-out* attribute of the target unit is not *null*, then the *int-in* read variable in the source gateway has to be T, but not vice versa, since *int-in* might be T due to some other unit in the subnet being ready to send a message. This is an example of an association relating the states of the participants in a way which is more sophisticated than shared memory. Of course, an eventual realisation of links would need to include elements for implementing this kind of more complex communication (a much more difficult task than implementing shared memory communication).

Let us postpone the definition of the semantics of associations to the next section.

1.3.8. *Specification of Complex Components*

Basic components are those whose internal state is not composed of other (simpler) components. Subsystems correspond to complex components, i.e., components whose internal structure is built out of the dynamic aggregation of interacting simpler components. With subsystem definitions we reach the upper layer of the language. Subsystems correspond to the concept of dynamic aggregation of components in architecture description languages. Subsystem definitions correspond to the definition of *composite components* in Darwin (although we do not allow for recursion in the definition of complex components), *systems* in Acme, dynamic configurations in CommUnity and *dynamic configurations* in Wright. Basically, a subsystem is composed, at a given instant in time, of a set of interacting instances of simpler components, which are related by instances of associations. Besides this "structural composition", subsystems might also contain basic attributes and read variables, which will allow it to communicate with other subsystems in the context of a bigger system. One can also restrict the visibility of a subsystem's attributes and operations, by means of an **Exports** clause; however, we will not make use of this facility in our examples. Again, the absence of of an **Exports** clause in a subsystem definition is interpreted as all the subsystem's attributes and operations being exported.

The most simple kind of subsystem, that we start describing now, is the one defined directly in terms of basic components. Subsystem *SubNet* is of this kind, and represents a dynamic collection of units connected to a gateway. A number of

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

requirements have already been made clear from the specification of the problem. Some of these are the following:

- There is always a non-empty set of units within the subnet,
- the units in the subnet are organised in a star topology, with a unit of a special kind, a gateway, in the centre of the star.

Note that, since the gateway must always be present in the subnet, the first of the above conditions is trivially met. We can make the "design decision" of representing the gateway as an attribute; in fact, this is somehow necessary, since we will need to relate some attributes, read variables and actions of the gateway with others outside the subnet, in order to enable communication between the subnet and other subsystems.

An extra datatype is necessary in order to start describing subsystems. This datatype is NAME, and represents the names of instances of components. According to the description of the problem, new units can be added to the subnet, and existing ones can be deleted (except for the gateway). So, we will need subsystem operations *add-unit*(NAME, address) (the second argument of the *add-unit*(NAME, address) operation will correspond to the address to be assigned to the new unit) and *rem-unit*(NAME) in order to characterise these tasks. Besides these two operations, we include an initialisation operation for the subsystem, called *s-init*(address) (the argument of the *s-init*(address) operation will correspond to the address to be assigned to the gateway of the subnet). A subsystem specification corresponding to the above description is shown in Fig. 1.5. Axioms 1-3 specify that the units in the subnet are arranged in a star topology, with *gw* in the centre. Axiom 4 indicates that attribute *gw* is, throughout the lifetime of the subnet, an instance of *Gateway*, and Axiom 5 says that *gw* does not change (while the subnet is active).

Axiom 6 corresponds to the informal requirement that an address must be unique to a unit[a]. It is worth noting that, thanks to inheritance, this requirement also involves the gateways in the subnet. Axiom 7 indicates that the addresses of the live units in the subnet respect the netmask specified within *gw*. This shows how the languages at different levels in the specification are related.

Axioms 8-9, 10-12 and 13-15 specify the behaviour associated with operations *s-init*(address), *add-unit*(NAME, address) and *rem-unit*(NAME), respectively. Axiom 16 complements the specification of association *Link*, by requiring the occurrence of the operation *gw.int-get*(message) to be associated with the occurrence of the operation $n.send(x)$, for some of the units linked to the gateway of the subnet.

The *SubNet* subsystem specification makes use of the language defined in simpler components *Unit* and *Gateway*, with a slight modification: an extra NAME-typed argument has been added to the flexible symbols originating in these components. Basically, if *at* is an attribute defined in class definition C, then symbol $at(n)$ 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

[a]Let us recall that this requirement was not specifiable within *Unit* or *Gateway*, since it is a structural constraint.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

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.

1.3.9. *Semantics of Subsystems*

As for class specifications, a subsystem *Sub* describes a theory presentation; its axioms are:

- The formulae explicitly provided in the subsystem specification,
- The (explicit and implicit) formulae corresponding to every class definition A aggregated by *Sub*, appropriately translated into the language of *Sub* by a relativisation,
- Implicit formulae characterising association definitions,
- Implicit formulae characterising general properties of subsystems.

1.3.9.1. *Relativisation of Classes*

The relativisation of class definitions is a simple procedure that transforms the axioms of the form α, given in a component definition C, into the property "all live instances of C have property α", for inclusion in a subsystem aggregating C. The activeness predicate of the corresponding class C is very important in this translation. Basically, the statement of the form "while the component is active, the property P holds" within a class C becomes the subsystem statement "for every component x, if x is a live instance of C, then x has property P".

The relativisation consists of the universal quantification of the extra argument of type NAME, introduced when mapping the language of a component into the language of an aggregating subsystem. As an example, let us consider the following axiom of class *Gateway*:

$$\Box[\mathit{Gateway} \rightarrow \mathit{netmask}(\mathit{addr})]$$

The relativisation of this axiom is the following:

$$\forall n \in \mathsf{NAME} : \Box[\mathit{Gateway}(n) \rightarrow n.\mathit{netmask}(n.\mathit{addr})]$$

Promoting Properties of Classes One of the advantages of our hierarchical organisation of reconfigurable systems in terms of simpler subsystems and classes is that we allow for further localisation of proofs to the relevant parts of a specification. For example, we can reason about some property concerning a class C within the theory of C, and then *promote* that property into an including subsystem. This is possible thanks to the fact that the logic admits a (derived) inference rule, that we call the *rule of structurality.* This rule asserts the following:

> If ψ is a consequence of a set Φ of premises, then the relativisation ψ' of ψ is a consequence of the relativisation Φ' of Φ.

This rule, and the fact that the relativisation of the axioms of classes are included in the theory of an aggregating subsystem, enable us to promote properties proved at the level of classes as properties of an including subsystem.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

We also need to provide a meaning to the flexible predicates representing the coverage of classes, related to class extension. Given a class definition C, predicate $\mathbf{C}$, used to denote the live instances of C or any subclass of C in a subsystem Sub, is simply defined by the following formula of the language of Sub:

$$\forall x : \mathbf{C}(x) \leftrightarrow (C(x) \vee C_1(x) \vee \ldots \vee C_k(x))$$

where $C_1, \ldots, C_k$ are the subclasses of C, i.e., all the classes which inherit, directly or indirectly, from C.

Polymorphic dynamic reconfiguration operations are easily defined, simply by using predicate $\mathbf{C}$ as an ordinary class predicate.

1.3.9.2. *Semantics of Associations*

From the connection definitions within associations, we generate some formulae, to be included in the theory of subsystems using the associations. In order to incorporate the formulae related 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 a binary association R (the generalisation of this for n-ary associations is trivial), where x and y are the free variables of $\alpha(x, y)$ representing the participants, then the formula:

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

is included in Sub, clearly characterising the fact that components referenced by x and y *must* colaborate according to the connections associated to R while they are "connected" by R (and while Sub is actively involved in the system).

We also indicate the "type" of each of the participants in an association definition. If R is an association definition with k participants, then for all i, with $1 \leq i \leq k$, if the i-th participant is defined to be of class C_i, then the following formula is incorporated in the theory of a subsystem Sub using R:

$$\Box[Sub \rightarrow [\forall x_1, \ldots, x_k : R(x_1, \ldots, x_k) \rightarrow C_i(x_i)]]$$

So, for our sample association $Link$, some of the formulae that will be implicitly included in the theory of $SubNet$ are the following:

$$\begin{array}{l}
\Box[SubNet \rightarrow [\forall s, t : Link(s, t) \rightarrow Gateway(s)]], \\
\Box[SubNet \rightarrow [\forall s, t : Link(s, t) \rightarrow \mathbf{Unit}(t)]], \\
\Box[SubNet \rightarrow [\forall s, t : Link(s, t) \rightarrow (t.curr\text{-}out \neq null) \rightarrow (s.int\text{-}in = \mathsf{T})]], \\
\Box[SubNet \rightarrow [\forall s, t : Link(s, t) \rightarrow (s.int\text{-}out = \mathsf{T}) \rightarrow (t.curr\text{-}in = null)]], \\
\vdots
\end{array}$$

1.3.9.3. *Other Properties of Subsystems*

There might be a number of other implicit axioms to include in a subsystem. These might be related to general assumptions or properties of the application domain. For instance, the locality of the subsystem [2] is generally assumed to be a property of subsystems, and therefore might be implicitly included in the theory describing

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

any subsystem. Axioms typing the associations can also be considered as general assumptions.

We consider for our sample specification three general assumptions easily expressible in the logic, namely: *(i)* that nothing can be at the same time an instance of two different classes, *(ii)* that operations of "dead" instances cannot take place, and *(iii)* that a subsystem may evolve only by means of its own operations (locality of a subsystem).

1.3.10. *Higher Level Subsystems*

Since the semantics of subsystems (i.e., components aggregations) is defined in terms of conventional temporal theories, as for classes, there is no technical restriction for iterating the process of defining aggregations. Therefore, we can permit subsystems not only to be composed of instances of classes, but also to subsume instances of simpler subsystems, thus allowing for hierarchical organisation of systems.

We have already defined several specification components, such as datatypes (particularly message and address), classes (*Unit* and *Gateway*), associations (*Link*) and a first subsystem (*SubNet*). We have made use of inheritance, in order to relate the definition of *Gateway* to *Unit*, and be able to define *Link* polymorphically. All these combined constitute the specification of a large component of the overall system, encapsulated in subsystem *SubNet*. We now want to build a bigger system combining different instances of interrelated subnets, as specified in the problem description.

Clearly, the problem description calls for the specification of a high level subsystem. Let us refer to it as *Net*. A net is composed of a dynamic set of subnets, all of which are connected to a gateway of the net, which, in order not to confuse it with the gateways of the subnets, we are going to call the *router*. Certainly, we will need reconfiguration operations in *Net* in order to manage the population of subnets within it.

Again, as we did with the *SubNet* subsystem, we make the design decision of representing the router as an attribute. This will allow us to make nets communicate with other nets in the environment (although we will not be concerned with the specification of this). The operations of a net are the following:

- An initialisation operation, *n-init*(address), whose argument will be used to set the address of the router in the net.
- An operation for adding new subnets, *add-subnet*(NAME, address), whose first argument represents the identifier of the subnet, and whose second argument represents the address that will be used to set the gateway of the new subnet.
- An operation for deleting existing subnets, *rem-subnet*(NAME), whose argument is the identifier of the live subnet to be removed from the net.

Relating the router to the live subnets within the net requires the definition of another association, this time a high level one. A problem arises because of our need for defining this association. Since *Gateway* is used within *SubNet*, the part of the language of *SubNet* which originated in *Gateway* (and has been relativised once because of this) *will be relativised again* when using it in a higher level subsystem,

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

say *Net*. Therefore, predicates such as *Gateway*(NAME) will be transformed by adding to them an extra NAME-typed argument; for instance, *Gateway*(NAME) will become *Gateway*(NAME, NAME), where the last argument refers to the subnet to which the gateway belongs.

But for the router, we simply want a gateway, not a gateway within the scope of any subnet. Note that we cannot directly use *Gateway*(NAME), since including the theory of *Gateway* relativised only once directly into the theory of *Net* will cause a clash of names between the doubly relativised *Gateway* (coming from *SubNet*) and the relativised *Gateway* (that we want to include directly into *Net*). In order to represent the router, we will use a renamed copy of *Gateway*. Let *Gateway'* be a theory presentation isomorphic to *Gateway*, obtained by "priming" the signature of *Gateway*, and appropriately translating the axioms.

The definition of association *S-Link*, whose purpose is to relate the router of a net with the gateways of the subnets within it, is shown in Fig. 1.6. Note how "multiple dots" in the dot notation we borrowed from object orientation are used. The intuitive reading is the top-down navigation from subsystem attributes towards their constituents. Recall that the multiple dots actually mean multiple NAME-typed arguments in the correponding expression. For example, the term *t.gw.curr-out* is a more readable version of *curr-out*(gw, t). This new NAME-typed parameter in the attributes, read variables and actions originating in *SubNet* correspond to a new relativisation, this time of the *SubNet* subsystem (which, we already know, contains the relativisation of the class definitions in the lower layer).

Note how similar the *S-Link* and *Link* associations are. Basically, the connections we employ in *S-Link* are the same as the ones in *Link*, except that this time we need to navigate into the t subnet to reach its gateway. Perhaps now it is clearer why we represented the gateway as a special attribute of *SubNet*, to facilitate the communication.

An informal graphical description of a particular state of a *Net* is shown in Fig. 1.7. We have depicted the router (labelled with `r1`) and five connected subnets, one of which shows its internal structure in terms of units. The *Net* specification is shown in Fig. 1.8. Subsystem *Net* has only one attribute, of type NAME, the router (rt). The router is meant to be a gateway (instance of *Gateway'*) through which the subnets within the net communicate with the outside world. The netmask of rt represents the netmask of the net, and therefore must subsume the netmasks of its live subnets. Axioms 1-3 and 8 specify how the subnets are connected to the router, again in a star topology. Also, they indicate that rt is the only instance of type *Gateway'*. Note that due to the types of the participants of association *S-Link* being different, we do not need an *anti-reflexivity* axiom (as we did in *SubNet*).

Axiom 4 indicates that the router does not change throughout the lifetime of the net. Axiom 5 says that the netmasks of different subnets within the net are "disjoint". This is a nice example of a formula relating instances at different layers of the language. Also, note the simplicity with which we have captured a rather complex constraint. Axiom 6 is another example of this kind. It specifies that the netmask of the router subsumes the netmasks of the live subnets within the net.

Axioms 7 and 9 specify the intended behaviour of the *n-init*(address) operation. Axioms 10-12 and 13-15 specify the intended behaviour of the operations that

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

manage the population of subnets in the net, i.e., *add-subnet*($\mathsf{NAME}, \mathsf{address}$) and *rem-subnet*($\mathsf{NAME}$). Note that Axioms 12 and 15 are stronger locality constraints. Finally, Axiom 16 complements the definition of association *S-Link*, by imposing that $rt.int\text{-}get'(x)$ cannot occur spontaneously, but only due to the occurrence of $n.gw.send(x)$, for some live subnet n linked to the router rt.

Let us summarise the theories involved in this specification, and how they are related. Besides the class definitions *Unit* and *Gateway*, a new actor is involved, namely *Gateway'*. The different relationships between theories in the specification of *Net* are shown in Fig. 1.9. Theory $\mathcal{ADT}_{\mathsf{NAME}}$ represents the conservative extension of $\mathcal{ADT}$ with sort NAME and a sufficiently large set of constants of this sort.

Arrows labelled with *id* indicate that the inclusion of the source theory into the target does not require any changes in the language of the source. Arrows labelled with *rel* indicate that the inclusion of the source theory into the target requires a relativisation of the language of the source. We can also see the double headed arrow relating *Gateway* and *Gateway'*, indicating the existence of an isomorphism between them (the translation involved is the "priming"). We can take advantage of the structurality rule, which allows us to translate proofs of theorems within a theory as proofs of their corresponding relativisations, in order to reuse reasoning, or reduce reasoning in some complex theories to reasoning within the smaller theories from which they are composed.

1.3.11. *Some Properties of Net*

We can employ the proof calculus for the logic in order to prove properties of the *Net* subsystem. Some of the properties that we have attempted to demonstrate are:

(1) "Different units do not share addresses":

$$\Box\,[\forall s_1, s_2, n_1, n_2 : Net \wedge \mathbf{Unit}(n_1, s_1) \wedge \mathbf{Unit}(n_2, s_2) \wedge (s_1 \neq s_2 \vee n_1 \neq n_2) \rightarrow s_1.n_1.addr \neq s_2.n_2.addr]$$

(2) "Gateways are initialised with valid addresses":

$$\Box[\forall s : \forall x \in \mathsf{address} : Net \wedge SubNet(s) \wedge s.gw.u\text{-}init(x) \rightarrow rt.netmask'(x)].$$

(3) "Messages are not lost":

$$\Box\,[\forall s_1, s_2, n_1, n_2 : \forall x \in \mathsf{message} : Net \wedge SubNet(s_1) \wedge SubNet(s_2) \wedge s_1.\mathbf{Unit}(n_1) \wedge s_2.\mathbf{Unit}(n_2) \wedge s_1.n_1.send(x) \wedge dest(x) = n_2 \rightarrow \Diamond s_2.n_2.get(x)]$$

The proofs of the first two are relatively straightforward, and relate reasoning at different levels in the specification. The third property is not valid in general, and various further assumptions have to be made in order to ensure it. The proof of the property can be modularised into various "lemmas", such as the following:

"Within a subnet, if a unit sends a message to another unit of the same subnet, the message is eventually received"

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

This property is characterised, within the language of subnets, as follows:

$$\Box\,[\forall n_1, n_2 : \forall x \in \mathsf{message} : \\ SubNet \wedge \mathbf{Unit}(n_1) \wedge \mathbf{Unit}(n_2) \wedge n_1.send(x) \wedge (dest(x) = n_2) \rightarrow \Diamond n_2.get(x)]$$

and can be proved straightforwardly by assuming the following extra constraints:

- n_2 is not gw; if n_2 is gw, then it will consume the message from its internal interface, i.e., via *int-get* rather than *get*,
- n_2 should not be deleted; it can be characterised easily by $\Box\mathbf{Unit}(n_2)$, (There exist less strong conditions in order to ensure that n_2 is not deleted before the message is received; we have chosen this one for the sake of simplicity, since it simplifies our proof.)
- the subnet should not be deleted; it can be characterised easily by $\Box SubNet$, (Again, there exist less strong conditions in order to ensure that the subnet is not deleted before the sent message is received; we have chosen this one for the sake of simplicity, since it simplifies our proof.)
- if the gateway gw holds a message addressed to an internal unit of the subnet, then that message is eventually dispatched; it can be characterised as follows:

  $$\Box[\forall x \in \mathsf{message} : SubNet \wedge (gw.int\text{-}curr\text{-}out = x) \wedge (x \neq null) \rightarrow \\ \Diamond gw.int\text{-}send(x)]$$

- if the gateway gw gets a message from its internal interface, addressed to its internal interface, then it eventually forwards it; it can be characterised as follows:

  $$\Box\,[\forall x \in \mathsf{message} : SubNet \wedge (gw.int\text{-}curr\text{-}in = x) \wedge (x \neq null) \wedge \\ gw.netmask(dest(x)) \wedge (dest(x) \neq gw.addr) \rightarrow \\ \Diamond(gw.int\text{-}prod(x)]$$

  Note that we have as part of the antecedent in this requirement the conjunct $(dest(x) \neq gw.addr)$, since if the destination of x is gw, the message should be consumed by gw instead of being forwarded; also, we have characterised the "forwarding" of the message as its production towards the internal interface.

1.4. Conclusions

We have presented a prototypical language for the specification of component based systems, with special support for reconfiguration. The way in which communication between components is achieved in the language (by means of dynamic connectors), standard in ADLs, allows it to express properties concerning the architecture of the system in a declarative way. Hence, operations that may change the topology of the system can be easily specified in the formalism. The presented prototypical specification language allows for the specification of reconfigurable component based systems by defining:

- basic datatypes, used then as types of *variables* of components,
- templates of basic components (components whose internal state is not composed of other components), called *class definitions*,

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

- templates of connectors, called *associations*,
- templates of complex components, whose internal state might be defined by a dynamic set of simpler components, interacting by means of connectors; both the population of components and the population of connectors can be dynamically changed in a subsystem, by means of reconfiguration operations.

The language also allows for the definition of inheritance relationships between component definitions. This is done for the purpose of defining polymorphic reconfiguration operations at the level of subsystems. Both the definitions of subsystems and associations are general enough to cope with complex components; subsystems can then be defined out of instances of simpler subsystems, similar to the situation in Darwin, although recursion is not allowed (due to the way a subsystem's semantics is defined); associations can also be defined to relate instances of subsystems.

The semantics of the language is defined in terms of a logic based on a suitable variant of an existing and widely known first-order linear temporal logic, the Manna-Pnueli logic. This variant generalises the symbols whose interpretation is state dependent (in the modal sense) to general predicate and function symbols. A sound proof calculus is available for this logic, and admits a derived proof rule that helps us in our proofs regarding reconfigurable component based systems.

We have been greatly motivated by the possibility of describing dynamic software architectures in a declarative way. We have attempted to maintain some interesting features of certain ADLs, particularly declarativeness (as in Acme), hierarchical composition of systems (as in Darwin) and designs at a high level of abstraction (as in CommUnity).

The main features the presented specification language exhibits are:

- A new modularity mechanism, the subsystem, which emerged as a consequence of our representation of component interaction.
- The use of inheritance and the associated polymorphism, attempting to mimic the situation of object oriented languages, in the context of software architectures. We have been careful not to bring into the language the complexities of fully fledged object oriented languages, and kept the organisation of specifications *hierarchical*.
- The use of classes and subsystems to build incremental specifications *hierarchically*. This was possible thanks to the representation of connections by means of coordination mechanisms, as in CommUnity.

Although we have not presented it in this article, we have defined a representation of our specifications as STeP [7] specifications. STeP is a tool for assistance in the specification and verification of concurrent programs based on the Manna-Pnueli logic. Although we have encoded the more general flexible symbols of the logic we use into Manna-Pnueli's, the use of the STeP tool for assistance in reasoning about our specifications (which involves these encodings) becomes unreasonably complicated for medium size specifications (as, for instance, our specification of nets). We are currently trying to overcome this difficulty, by incorporating support for general flexible symbols and the rule of structurality into the STeP tool.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

We are planning to work on the development of an ADL using the ideas presented, and trying to learn from the positive and negative aspects of other existing ADLs and environments for software architectures.

Bibliography

1. M. Abadi, *The Power of Temporal Proofs*, Theoretical Computer Science 65:35-84.
2. 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.
3. R. Allen, R. Douence and D. Garlan, *Specifying and Analyzing Dynamic Software Architectures*, in Proceedings of Fundamental Approaches to Software Engineering FASE 98, Lisbon, Portugal, Lecture Notes in Computer Science, Springer-Verlag, 1998.
4. R. Allen and D. Garlan, *Formalizing Architectural Connection*, in Proceedings ICSE '94, Sorrento, Italy, 1994.
5. L. Andrade and J. Fiadeiro, *Interconnecting Objects via Contracts*, in UML'99 -Beyond the Standard, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag, 1999.
6. L. Andrade and J. Fiadeiro, *Service-oriented Business and System Specification: Beyond Object-orientation*, in Practical Foundations of Business System Specifications, H. Kilov (ed.), Kluwer Academic Publishers, 2003.
7. N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, M. Pichora, H. Sipma and T. Uribe, *STeP, The Stanford Temporal Prover*, User's Manual, Computer Science Department, Stanford University, 1998.
8. C. Britton, *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems*, Addison-Wesley, 2000.
9. S. Cook and J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, The Object-Oriented Series, Prentice-Hall, 1994.
10. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of EATCS Monographs on Theoretical Computer Science, Springer, 1985.
11. J. Fiadeiro and T. Maibaum, *Temporal Theories as Modularisation Units for Concurrent System Specification*. Formal Aspects of Computing, vol. 4, No. 3, Springer, 1992.
12. J. Fiadeiro and T. Maibaum, *Sometimes "Tomorrow" is "Sometime", Action Refinement in a Temporal Logic of Objects*, in D. Gabbay and H. Ohlbach (eds), Temporal Logic, LNAI 827, Springer-Verlag, 1994.
13. J. Fiadeiro and T. Maibaum, *Categorical Semantics of Parallel Program Design*, Science of Computer Programming 28(2-3), 1997.
14. E. Gamma, R. Helm, R. Johnson and j. Vlissides, *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley, 1994.
15. D. Garlan, *Software Architecture: A Roadmap*, in The Future of Software Engineering, A. Filkenstein (ed), ACM Press, 2000.
16. D Garlan, R. Monroe and D Wile, *ACME: An Architecture Description Interchange Language*, in Proc. of CASCON'97, 1997.
17. D. Garlan, R. Monroe and D Wile, *ACME: Architectural Description of Component-Based Systems*, in Foundations of Component-Based Systems, Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, 2000.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

18. P. Inverardi and A. Wolf, *Formal Specification and Analysis of Software Architetures using the Chemical Abstract Machine*, IEEE Transactions in Software Engineering, 1995.
19. J. Magee and J. Kramer, *Dynamic Structure in Software Architectures*, in Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, California, USA, 1996.
20. 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.
21. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.
22. N. Medvidovic, *ADLs and Dynamic Architecture Changes*, in Proceedings of the Second Int. Software Architecture Workshop (ISAW-2), 1996.
23. N. Medvidovic and R. Taylor, *A Framework for Classifying and Comparing Architecture Description Languages*, In ESEC-FSE'97, 1997.
24. D. Parnas, *A Technique for Software Module Specification with Examples*, in Communications of the ACM, 15(5), 1972.
25. D. Parnas, *On the Criteria to be Used in Decomposing Systems into Modules*, in Communications of the ACM, 15(12), 1972.
26. J. Rumbaugh, *Relations as Semantic Constructs in an Object-Oriented Language*. In Proceedings of OOPSLA '87, Orlando, USA, 1987.
27. M. Wermelinger, A. Lopes and J. Fiadeiro, *A Graph Based Architectural (Re)configuration Language*, in ESEC/FSE'01, V.Gruhn (ed), ACM Press, 2001.
28. M. Wermelinger and J. Fiadeiro, *A Graph Transformation Approach to Software Architecture Reconfiguration*, in Science of Computer Programming 44, Elsevier, 2002.

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

Class *Unit*
Exports *curr-in*, *curr-out*, *u-init*(address),
prod(message), *send*(message), *get*(message), *cons*(message), *rem*()
Read Variables

in : boolean
out : boolean

Attributes

addr : address
curr-in : message
curr-out : message

Actions

u-init(address)
prod(message)
send(message)
get(message)
cons(message)
rem()

Axioms

(1) $\Box[\forall x \in \text{address} :$
$Unit \wedge u\text{-}init(x) \rightarrow \bigcirc((addr = x) \wedge (curr\text{-}in = null) \wedge (curr\text{-}out = null))]$

(2) $\Box[\forall x \in \text{address} :$
$Unit \wedge u\text{-}init(x) \rightarrow \bigcirc(\neg(\exists y \in \text{address} : u\text{-}init(y))\mathcal{W}\neg Unit)]$

(3) $\Box[Unit \wedge (addr \neq \bigcirc addr) \rightarrow \exists x \in \text{address} : u\text{-}init(x)]$

(4) $\Box[\forall x \in \text{message} : Unit \wedge get(x) \rightarrow ((in = \mathsf{T}) \wedge (curr\text{-}in = null))]$

(5) $\Box[\forall x \in \text{message} : Unit \wedge get(x) \rightarrow \bigcirc(curr\text{-}in = x)]$

(6) $\Box[\forall x \in \text{message} : Unit \wedge cons(x) \rightarrow$
$((curr\text{-}in \neq null) \wedge (curr\text{-}in = x) \wedge (dest(x) = addr))]$

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

(8) $\Box[Unit \wedge rem() \rightarrow ((curr\text{-}in \neq null) \wedge (dest(curr\text{-}in) \neq addr))]$

(9) $\Box[Unit \wedge rem() \rightarrow \bigcirc(curr\text{-}in = null)]$

(10) $\Box[Unit \wedge (curr\text{-}in \neq \bigcirc curr\text{-}in) \rightarrow (\exists x \in \text{message} : get(x) \vee cons(x)) \vee rem()]$

(11) $\Box[\forall x \in \text{message} : Unit \wedge send(x) \rightarrow ((out = \mathsf{T}) \wedge (curr\text{-}out = x))]$

(12) $\Box[\forall x \in \text{message} : Unit \wedge send(x) \rightarrow \bigcirc(curr\text{-}out = null)]$

(13) $\Box[\forall x \in \text{message} : Unit \wedge prod(x) \rightarrow (curr\text{-}out = null)]$

(14) $\Box[\forall x \in \text{message} : Unit \wedge prod(x) \rightarrow \bigcirc(curr\text{-}out = x)]$

(15) $\Box[Unit \wedge (curr\text{-}out \neq \bigcirc curr\text{-}out) \rightarrow \exists x \in \text{message} : send(x) \vee prod(x)]$

EndofClass

Fig. 1.2. Class *Unit*: A specification of a component that interchanges messages

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

Class *Gateway*
Extends *Unit*
Exports *curr-in*, *curr-out*, *int-curr-in*, *int-curr-out*, *u-init*(address), *prod*(message), *send*(message), *get*(message), *cons*(message), *rem*() *int-prod*(message), *int-send*(message), *int-get*(message), *int-cons*(message), *int-rem*()
Read Variables *int-in* : boolean, *int-out* : boolean
Attributes *netmask* : address → boolean, *int-curr-in* : message, *int-curr-out* : message
Actions

int-prod(message), *int-send*(message), *int-get*(message), *int-cons*(message), *int-rem*()

Axioms

(1) $\Box[Gateway \rightarrow netmask(addr)]$

(2) $\Box[\forall x \in \mathsf{address} : netmask(x) \leftrightarrow \bigcirc netmask(x)]$

(3) $\Box[\forall x \in \mathsf{message} : Gateway \wedge int\text{-}get(x) \rightarrow ((int\text{-}in = \mathsf{T}) \wedge (int\text{-}curr\text{-}in = null))]$

(4) $\Box[\forall x \in \mathsf{message} : Gateway \wedge int\text{-}get(x) \rightarrow \bigcirc(int\text{-}curr\text{-}in = x)]$

(5) $\Box[\forall x \in \mathsf{message} : Gateway \wedge int\text{-}cons(x) \rightarrow ((int\text{-}curr\text{-}in \neq null) \wedge (int\text{-}curr\text{-}in = x) \wedge (dest(x) = address))]$

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

(7) $\Box[Gateway \wedge int\text{-}rem() \rightarrow ((int\text{-}curr\text{-}in \neq null) \wedge (dest(int\text{-}curr\text{-}in) \neq address))]$

(8) $\Box[Gateway \wedge int\text{-}rem() \rightarrow \bigcirc(int\text{-}curr\text{-}in = null)]$

(9) $\Box[Gateway \wedge (int\text{-}curr\text{-}in \neq \bigcirc int\text{-}curr\text{-}in) \rightarrow (\exists x \in \mathsf{message} : int\text{-}get(x) \vee int\text{-}cons(x)) \vee int\text{-}rem()]$

(10) $\Box[\forall x \in \mathsf{message} : Gateway \wedge int\text{-}send(x) \rightarrow ((int\text{-}out = \mathsf{T}) \wedge (int\text{-}curr\text{-}out = x))]$

(11) $\Box[\forall x \in \mathsf{message} : Gateway \wedge int\text{-}send(x) \rightarrow \bigcirc(int\text{-}curr\text{-}out = null)]$

(12) $\Box[\forall x \in \mathsf{message} : Gateway \wedge int\text{-}prod(x) \rightarrow (int\text{-}curr\text{-}out = null)]$

(13) $\Box[\forall x \in \mathsf{message} : Gateway \wedge int\text{-}prod(x) \rightarrow \bigcirc(int\text{-}curr\text{-}out = x)]$

(14) $\Box[Gateway \wedge (int\text{-}curr\text{-}out \neq \bigcirc int\text{-}curr\text{-}out) \rightarrow \exists x \in \mathsf{message} : int\text{-}send(x) \vee int\text{-}prod(x)]$

(15) $\Box[Gateway \wedge rem() \wedge netmask(dest(curr\text{-}in)) \rightarrow int\text{-}prod(curr\text{-}in)]$

(16) $\Box[Gateway \wedge int\text{-}rem() \wedge netmask(dest(int\text{-}curr\text{-}in)) \rightarrow int\text{-}prod(int\text{-}curr\text{-}in)]$

(17) $\Box[Gateway \wedge int\text{-}rem() \wedge \neg netmask(dest(int\text{-}curr\text{-}in)) \rightarrow prod(int\text{-}curr\text{-}in)]$

EndofClass

Fig. 1.3. Class *Gateway*: A specification of a component that forwards messages

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

Association *Link*
Participants

s : *Gateway*
t : **Unit**

Connections

$(t.curr\text{-}out \neq null) \rightarrow (s.int\text{-}in = \mathsf{T})$
$(s.int\text{-}out = \mathsf{T}) \rightarrow (t.curr\text{-}in = null)$
$(t.in = \mathsf{T}) \leftrightarrow (s.int\text{-}curr\text{-}out \neq null)$
$(t.out = \mathsf{T}) \leftrightarrow (s.int\text{-}curr\text{-}in = null)$
$\forall m \in \mathsf{message} : (s.int\text{-}send(m) \rightarrow t.get(m))$
$\forall m \in \mathsf{message} : (t.send(m) \rightarrow s.int\text{-}get(m))$

EndofAssociation

Fig. 1.4. An association defined to enable communication between gateways and units within subnets

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

Subsystem *SubNet*
Attributes

gw: NAME

Operations

$s\text{-}init(\mathsf{address})$
$add\text{-}unit(\mathsf{NAME}, \mathsf{address})$
$rem\text{-}unit(\mathsf{NAME})$

Axioms

(1) $\Box[\forall n, m : SubNet \wedge Link(n, m) \rightarrow (n = gw)]$

(2) $\Box[\forall n : SubNet \wedge \mathbf{Unit}(n) \wedge (n \neq gw) \rightarrow Link(gw, n)]$

(3) $\Box[SubNet \rightarrow \forall n : \neg Link(n, n)]$

(4) $\Box[SubNet \rightarrow Gateway(gw)]$

(5) $\Box[\forall n : SubNet \wedge (gw = n) \rightarrow ((gw = n)\mathcal{W}\neg SubNet)]$

(6) $\Box[\forall n, m : SubNet \wedge \mathbf{Unit}(n) \wedge \mathbf{Unit}(m) \wedge (n \neq m) \rightarrow (n.addr \neq m.addr)]$

(7) $\Box[\forall n : SubNet \wedge \mathbf{Unit}(n) \rightarrow gw.netmask(n.addr)]$

(8) $\Box[\forall x \in \mathsf{address} : SubNet \wedge s\text{-}init(x) \rightarrow gw.u\text{-}init(x)]$

(9) $\Box[\forall x \in \mathsf{address} : SubNet \wedge s\text{-}init(x) \rightarrow \bigcirc(\forall n : \mathbf{Unit}(n) \rightarrow (n = gw))]$

(10) $\Box[\forall n : \forall x \in \mathsf{address} : SubNet \wedge add\text{-}unit(n, x) \rightarrow \neg\mathbf{Unit}(n)]$

(11) $\Box[\forall n : \forall x \in \mathsf{address} : SubNet \wedge add\text{-}unit(n, x) \rightarrow \bigcirc(\mathbf{Unit}(n) \wedge n.u\text{-}init(x))]$

(12) $\Box[\forall n : SubNet \wedge (n \neq gw) \wedge \neg\mathbf{Unit}(n) \wedge \bigcirc(\mathbf{Unit}(n)) \rightarrow \exists x \in \mathsf{address} : add\text{-}unit(n, x)]$

(13) $\Box[\forall n : SubNet \wedge rem\text{-}unit(n) \rightarrow \mathbf{Unit}(n)]$

(14) $\Box[\forall n : SubNet \wedge rem\text{-}unit(n) \rightarrow \bigcirc(\neg\mathbf{Unit}(n))]$

(15) $\Box[\forall n : SubNet \wedge \mathbf{Unit}(n) \wedge \bigcirc(\neg\mathbf{Unit}(n)) \rightarrow rem\text{-}unit(n)]$

(16) $\Box[\forall x \in \mathsf{message} : SubNet \wedge gw.int\text{-}get(x) \rightarrow \exists n : Link(gw, n) \wedge n.send(x)]$

EndofSubsystem

Fig. 1.5. *SubNet*: A subsystem specification aggregating units and gateways

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

Association *S-Link*
Participants

s : *Gateway'*
t : *SubNet*

Connections

$(t.gw.curr\text{-}out \neq null) \rightarrow (s.int\text{-}in' = \mathsf{T})$
$(s.int\text{-}out' = \mathsf{T}) \rightarrow (t.gw.curr\text{-}in = null)$
$(t.gw.in = \mathsf{T}) \leftrightarrow (s.int\text{-}curr\text{-}out' \neq null)$
$(t.gw.out = \mathsf{T}) \leftrightarrow (s.int\text{-}curr\text{-}in' = null)$
$\forall m \in \mathsf{message} : (s.int\text{-}send'(m) \rightarrow t.gw.get(m))$
$\forall m \in \mathsf{message} : (t.gw.send(m) \rightarrow s.int\text{-}get'(m))$

EndofAssociation

Fig. 1.6. An association defined for communication between routers with subnets

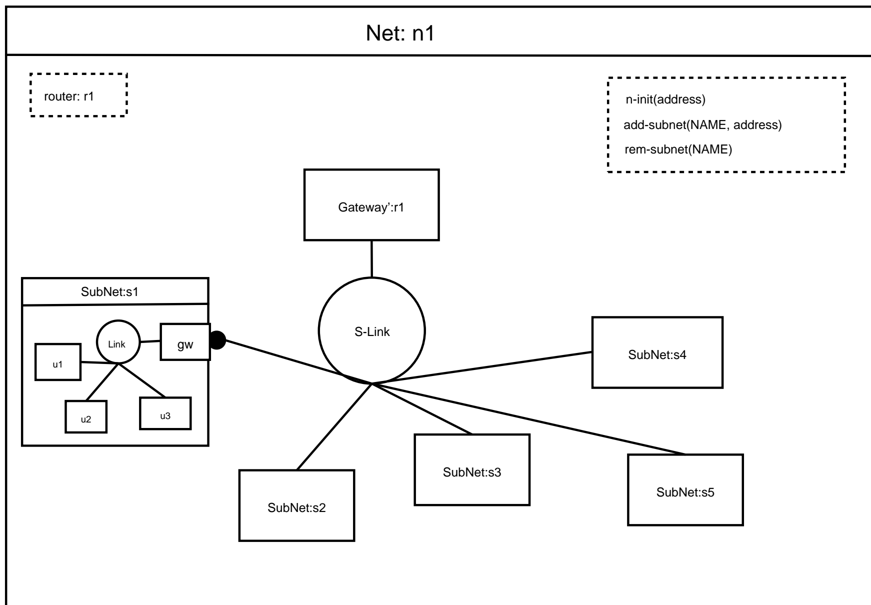

Fig. 1.7. Graphical representation of a configuration of subsystem *Net*

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

Subsystem *Net*
Attributes

rt: NAME

Operations

$n\text{-}init(\mathsf{address})$
$add\text{-}subnet(\mathsf{NAME}, \mathsf{address})$
$rem\text{-}subnet(\mathsf{NAME})$

Axioms

(1) $\Box[\forall n, m : Net \wedge S\text{-}Link(n, m) \rightarrow (n = rt)]$

(2) $\Box[\forall n : Net \wedge SubNet(n) \rightarrow S\text{-}Link(rt, n)]$

(3) $\Box[Net \rightarrow Gateway'(rt)]$

(4) $\Box[\forall n : Net \wedge (rt = n) \rightarrow ((rt = n)\mathcal{W}\neg Net)]$

(5) $\Box[\forall n, m : Net \wedge SubNet(n) \wedge SubNet(m) \wedge (n \neq m) \rightarrow (\forall x \in \mathsf{address} : \neg(n.gw.netmask(x) \wedge m.gw.netmask(x)))]$

(6) $\Box[\forall n : Net \wedge SubNet(n) \rightarrow (\forall x \in \mathsf{address} : n.gw.netmask(x) \rightarrow rt.netmask'(x)]$

(7) $\Box[\forall x \in \mathsf{address} : Net \wedge n\text{-}init(x) \rightarrow rt.u\text{-}init'(x)]$

(8) $\Box[Net \rightarrow (\forall n : Gateway'(n) \leftrightarrow (n = rt))]$

(9) $\Box[\forall x \in \mathsf{address} : Net \wedge n\text{-}init(x) \rightarrow \bigcirc(\forall n : \neg SubNet(n))]$

(10) $\Box[\forall n : \forall x \in \mathsf{address} : Net \wedge add\text{-}subnet(n, x) \rightarrow \neg SubNet(n)]$

(11) $\Box[\forall n : \forall x \in \mathsf{address} : Net \wedge add\text{-}subnet(n, x) \rightarrow \bigcirc(SubNet(n) \wedge n.s\text{-}init(x))]$

(12) $\Box[\forall n : Net \wedge \neg SubNet(n) \wedge \bigcirc(SubNet(n)) \rightarrow \exists x \in \mathsf{address} : add\text{-}subnet(n, x)]$

(13) $\Box[\forall n : Net \wedge rem\text{-}subnet(n) \rightarrow SubNet(n)]$

(14) $\Box[\forall n : Net \wedge rem\text{-}subnet(n) \rightarrow \bigcirc(\neg SubNet(n))]$

(15) $\Box[\forall n : Net \wedge SubNet(n) \wedge \bigcirc(\neg SubNet(n)) \rightarrow rem\text{-}subnet(n)]$

(16) $\Box[\forall x \in \mathsf{message} : Net \wedge rt.int\text{-}get'(x) \rightarrow \exists n : S\text{-}Link(rt, n) \wedge n.gw.send(x)]$

EndofSubsystem

Fig. 1.8. *Net*: A high level subsystem specification aggregating instances of subsystem *SubNet*

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html

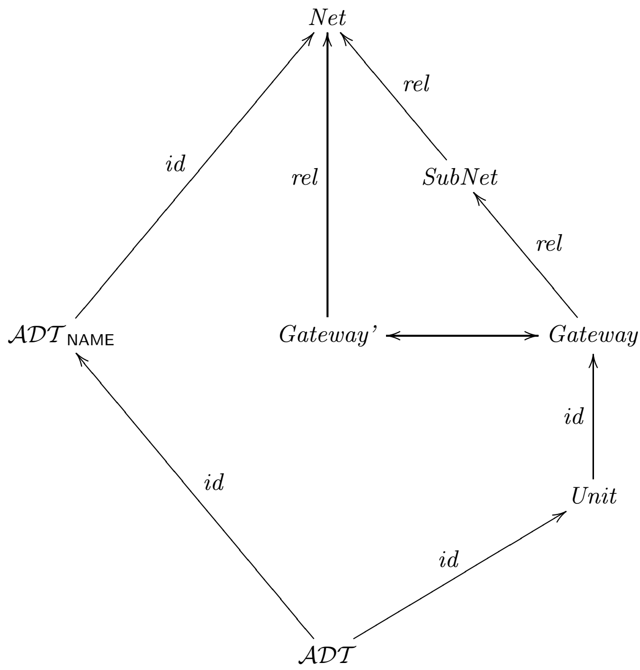

Fig. 1.9. Relationships among theories of the specification of *Net*

MATHEMATICAL FRAMEWORKS FOR COMPONENT SOFTWARE - Models for Analysis and Synthesis
© World Scientific Publishing Co. Pte. Ltd.
http://www.worldscibooks.com/compsci/6231.html