Architecture-Driven Verification of Concurrent Systems

Erdogmus, Hakan

*Architecture-driven verfication of concurrent systems.**

Erdogmus, H.

1997

* published in: Nordic Journal of Computing. 4: 380-413; 1997. NRC 41549.

Nordic Journal of Computing

Architecture-Driven Verification of Concurrent Systems

Hakan Erdogmus
National Research Council of Canada
Software Engineering Group
Building M-50, Montreal Road
Ottawa, Ontario, Canada K1A 0R6
`erdogmus@iit.nrc.ca`
`wwwsel.iit.nrc.ca`

Abstract. This paper proposes a method to construct a set of proof obligations from the architectural specification of a concurrent system. The architectural specifications used express correctness requirements of a concurrent system at a high level without any reference to component functionality. Then the proof obligations derived from such specifications are discharged as model checking tasks in a suitable behavioral model where components are assigned their respective functionalities. An experimental extension to the SPIN tool is used as the model checker. The block diagram notation used to specify architectures allows interchangeable components with equivalent intended functionalities to be encapsulated within a representative module. A proof obligation of such a system is discharged as an equivalence checking task in the behavioral model chosen. It is shown how infeasible proof obligations can be decomposed by decomposing the architectural specification. Obligation decomposition relies on assume-guarantee conditions.

Key words: architecture-based verification, architectural specifications, architectural formalisms, model checking, equivalence checking, compositional verification.

CR Classification: D.2.1[**Software Engineering**]:Requirements/Specifications; D.2.4[**Software Engineering**]:Program Verification; F.3.1[**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs; C.2.4[**Computer-Communication Networks**]: Distributed Systems.

1. Introduction

Architectural specifications have an increasingly important role in the design and development of computer-based systems. This role is evidenced by the emergence of software architecture with its formal underpinnings as a distinct field [28,50,55,3,18,13], and by the proliferation of specialized methods and notations for architectural descriptions of software [6,14,27,43,53,26]. Besides their use as documentation and as an abstraction mechanism to manage structural complexity, architectural specifications can also aid in rigorous system development based on step-wise refinement [49,52], in rapid system prototyping [52,36,51,43], in pre-implementation analysis [48,34,42,5], and in system development through reuse and instantiation of reference models [54,47,7,57,26,53].

Received July, 1996. Revised April, September 1997.

This paper addresses architectural specifications in terms of their ability to express correctness requirements of concurrent systems: architecture is exploited for verification. A method is proposed to construct a set of proof obligations from an architectural specification of a concurrent system. It is also shown how infeasible obligations can be decomposed by decomposing the architectural specifications themselves. Once the proof obligations are determined, each obligation can be discharged as a model checking task in a chosen behavioral model. The central idea of the paper is that proof obligations can be derived from structure and that decomposition of structure leads to decomposition of obligations. The criteria for selecting proof obligations from a redundant set are touched upon here, and discussed in more detail in [22] where a simple algorithm is provided to remove redundancy from a given set of obligations.

The architecture of a concurrent system is often expressed in terms of block diagrams with boxes and lines. Boxes represent modules, and lines represent connections. A diagrammatic notation of this type is used in this paper. Since such a notation alone does not address the functionality of the systems specified, modules are assigned their respective functionalities in some suitable behavioral model. Provided the behavioral model has a notion of composition of its own, it is necessary to assign functionality only to primitive modules which are found at the lowest level of an architectural specification.

Proof obligations are derived from modules which possess interchangeable components. These modules are called *functional groups*: a functional group represents a set of modules with equivalent intended functionalities. Typically, a functional group involves an abstract module and a number of concrete modules which represent alternative decompositions of the abstract module. If modules M and N belong to the same functional group F, then $\{M, N\}$ is a proof obligation of F. Such an obligation is discharged by checking $\beta(M) \equiv \beta(N)$, where β associates a corresponding functionality, in a given behavioral model, with each module; and $\equiv$ is an equivalence relation on behaviors. The proof obligations of a system are independent of the actual behavioral model and the equivalence relation adopted.

The problem of deciding behavioral relations between system specifications is not addressed here. This problem is well known in the model checking literature; the reader is referred to [19,11,40,25,56]. Examples of behavioral equivalences can be found in [17,16,39,46,30].

The distinction between open and closed systems is important in the proposed approach. An open system can interact with other systems, whereas a closed system can only be observed. It is assumed that only proof obligations of closed systems can be discharged. Thus $\{M, N\}$ cannot be discharged if M and N are open. However, if M and N are embedded in a closed context C as $C[M]$ and $C[N]$, respectively, then $\{C[M], C[N]\}$ may constitute a legitimate proof obligation that can be discharged. The context C models the common environment of M and N; it captures the assumptions that apply to the potential users of these two modules. Note that while some model checkers such as CÆSAR-ALDÉBARAN [25] can handle open systems, others such as SPIN [33] can not. The idea that it is possible to reason about open systems using explicit environmental assumptions and established methods for reasoning about closed systems has been suggested in [2].

The paper is organized as follows:

Section 1.1 reviews the related literature on architectural notations, compositional verification of concurrent systems, and architecture-driven verification. This is followed in Section 2 by the introduction of the model and the corresponding graphical notation used for architectural specifications. Section 3 introduces the formal model: module systems. The subject of Section 4 is the expression and derivation of correctness requirements in the architectural model. The concept of proof obligation is formalized here. Section 5 deals with obligation decomposition. Two kinds of decomposition techniques are presented: horizontal decomposition and vertical decomposition. A case study is discussed in Section 5.3 to illustrate these two techniques. Section 6 discusses the specification and verification of system architectures using PROMELA and SPIN. The paper is concluded with a discussion in Section 7.

1.1 Related Work

The use of a formalism based on box-and-line diagrams to describe system architectures is not novel. Neither is the idea of expressing component (and system) variability in terms of groups of interchangeable modules.

Numerous techniques have been proposed for architectural descriptions. Examples include specification languages which support a box-and-line type block diagram notation, such as ROOM [52], SDL [51,35], and ModeChart [36]; architectural description languages such as Aesop [26], Wright [6], UniCon [53], Rapide [43], and ACME [27]; and other formal notations such as those used in [49], [31], and [18]. All of these techniques refer to the notions of component, connector, and configuration to describe interface-connection architectures (systems of modules interconnected through their interface ports) [44].

While the separation of structure from behavior (functionality) is evident in many of these techniques [18,53,27,52,36,51], some formal approaches to software architecture [49,42,4,34] augment structural descriptions with behavioral descriptions of interactions between system components. This mixing of structure and functionality allows architectural specifications to be animated, and also certain types of behavioral analyses to be performed on them. As opposed to these latter methods, in this paper architecture refers exclusively to structure: hence structural and behavioral models are separated.

A central feature of the proposed architectural model is its ability to express families of systems (variability) in terms of module groups. A similar idea has been suggested in [7] and [57]. [7] uses realms and type equations to represent a system family. A realm is akin to a set of interchangeable, singular modules which implement the same interface (analogous to a union here). A type equation defines a module group and uses realms as components. In [57], a system family is made up of different versions of a generic system where each version structurally differs from the others in some way. Variability is expressed by conditional expressions which specify where and which variable components are included in different versions.

Architecture-based verification has been addressed in [48], [42], [4], and [34]. All of these works focus on interactions between components as the basis of verification, whereas here the focus is on the system structure itself. [48] and [4] employ a notation based on the familiar component-connector-configuration paradigm to specify structure: [48] uses a logical formalism for connector semantics while [4] uses a process algebraic one. [42] proposes an event-based, object-oriented formalism to specify executable architectures; structure is denoted again in terms of components, connectors, and configurations. This latter formalism is considerably more expressive than those of [48] and [4]. [48] and [42] attack the problem of deciding whether a concrete architecture conforms to, or implements an abstract architecture. While the underlying reasoning is purely behavioral and conformance is checked through animation in [42], [48] takes into account structural properties, defines correctness criteria which give rise to proof obligations, and uses logic-based proofs. By contrast, [4] deals with correctness properties within a single architecture: proof obligations are generated to ensure components interact in desirable ways. The approach of [34] is different in that it uses an operational model to specify high-level interactions between components. Structure is not explicitly specified: instead, it results from the underlying behavior. [34] also uses formal proofs to check specific behavioral properties of a single architecture. Unlike all of these works, here proof obligations are derived from purely structural specifications although they are discharged in a behavioral model. The choice of the behavioral model and the correctness criterion is independent of the structural specification. Therefore, the correctness criterion can easily be changed depending on which properties should be preserved, without affecting the structural specification.

Compositional verification of concurrent systems have been explored in many articles [45,37,41,15,38,1,2,12,29]. In [45], [37], [41] and [38], as well as here, correctness is defined in terms of a behavioral relation. In the remaining references, correctness is defined in terms of properties expressed in a modal logic. [45] uses abstractions of different levels as the basis of decomposition. In [37], a system satisfies its specification if its components satisfy their respective specifications. In [41, Ch. 6], a property holds for a system defined in terms of a conjunction of constraints if it holds for each constraint individually.

DeLeon and Grumberg describe a compositional method based on reduction for temporal model checking in [15]. To verify that a composite system satisfies a given property, each component is reduced by abstracting away details irrelevant for the property to be verified. The resulting abstractions are then recomposed, and the property is verified in this reduced system.

This idea is similar to the decomposition technique (vertical decomposition) discussed in Section 5.2: a composite system is verified using abstractions of its components.

Similarly, Kurshan describes a compositional verification method based on homomorphic reduction for testing ω-language inclusion between two concurrent systems [38]. The notion of reduction described in [38] and [15] is a powerful one in that it is defined relative to the property to be proved. Clarke and others [12] propose a technique to compositionally obtain such reductions (called approximations therein) directly from the syntactic description of a concurrent program. Their technique is constructive in that the validity of the reductions does not have to be checked. Kurshan's notion of reduction and the notion of approximation defined by Clarke and others are analogous to the notion of approximation discussed in Section 4, where approximations apply to closed contexts and are defined relative to a system which can be enclosed in those contexts. By contrast in [38] and [12], reductions or approximations apply to systems and are defined relative to a behavioral property.

In [29], Grumberg and Long describe another compositional verification method for temporal model checking. This method requires environmental assumptions expressed in an assume-guarantee style [2] to be explicitly specified in order to reason about an open system. To prove that a component (subsystem) enclosed in a particular context (the environment) satisfies a given property, first the context is abstracted in a way independent of the property to be proved. Then the property is verified with the component enclosed in the abstracted context. The abstracted context captures the assumptions regarding the environment. Unlike in [12], it is not described how the abstract context can be obtained: thus the validity of the abstraction needs to be checked explicitly. Note that the notion of abstraction is not as powerful as that of reduction or approximation because rather than being defined relative to a particular property, it is defined relative to a large set of properties. The method of Section 5 for obligation decomposition is also based on an assume-guarantee type reasoning: environmental assumptions are expressed as approximation relationships among contexts.

2. Representation of System Architectures

Here system architectures are represented graphically as block diagrams. These diagrams consist of nested boxes, called *modules*, which can be interconnected in different ways. A module may denote a single system or a class of systems.

Fig. 1 summarizes the relationships between the different kinds of modules and introduces the graphical notation used. The arrows represent an inheritance-like relationship. Meeting points of arrows imply exclusive disjunction. For example, a primitive module *is a* singular module, which in turn *is an* interaction module; a group *is either a* composite *or a* union; a

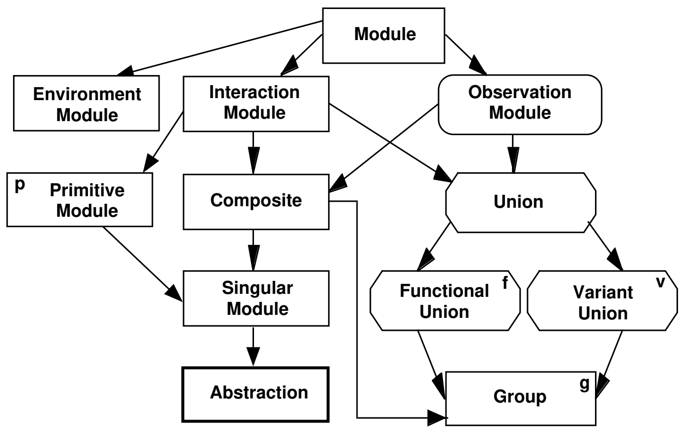


Fig. 1: Module terminology and notation.

union *is either a* functional union *or a* variant union; an abstraction *is a* singular module; and so on.

The exposition of this section is mostly informal. The underlying formal model of modules is defined in Section 3. A more elaborate model can be found in [20].

2.1 Designation of a Module

The *designation* of a module specifies the purpose of the module. Three designations are possible: *observation*, *interaction*, and *environment*.

2.1.1 Observation Modules and Interaction Modules

An *observation module* (drawn as a box with rounded corners) represents a closed system. Such a system is self-contained: it may be observed, but it does not interact with the observer or other modules [2]. Therefore, observation modules are found only at the top layer of the module hierarchy of an architectural specification.

An *interaction module* (drawn as a rectangular box) represents an open subsystem. A module of this kind can be understood as a *type* whose instances can serve as components in larger systems, or more precisely, in composite modules.

2.1.2 Environment Module

An *environment* module (drawn as a small square) is like an interaction module, but unlike that of an interaction module, its interface is not explicitly specified. Environment modules represent shared resources, and are composed typically of variable declarations and access macros. An environment module has no structure defined for it, and as such, is always found at the bottom layer of a module hierarchy. Environment modules are *owned* as components by composite modules. An environment module can not be a component of (owned by) more than one composite.

2.2 Interface of a Module

Every interaction or observation module has a nonempty *interface* which consists of a finite set of *ports*. The interface of a module M is denoted by $\mathcal{I}_M$.

In an observation module, the interface exposes the module's relevant internal behavior to be observed by an external agent. By observing the interface, the external agent can reason about the system described by the module. For an interaction module, the interface specifies the boundary through which instances of that module can be interconnected with instances of the same or other modules within a larger, composite module.

2.3 Structural Type of a Module

Independent of their designation, interaction and observation modules are classified according to their internal structure. The *structural type* of a module can be *primitive*, *composite*, or *union*.

2.3.1 Primitive Modules

A *primitive module* (marked by a "**p**" in block diagrams) is an interaction module with no further internal structure. It represents an elementary subsystem, and like an environment module, occupies a leaf node in the module hierarchy of an architectural specification. The structure of a primitive module consists only of an interface. Observation modules cannot be primitive.

2.3.2 Composites

A *composite module*, or simply *composite*, (drawn as a rectangular box) is comprised of a network of interconnected *components*. Therefore the structure of a composite consists of an interface, a set of components, and a set of connections. Composites occupy nonleaf nodes in a module hierarchy.

A composite possesses at least one component. Each component is either an environment module or an *instance* of some interaction module. For a component x which is an instance of an interaction module I, the notation x_I is used. Then I is called the *type* of x. The statements "x is a component of type I of M" and "x_I is a component of M" are equivalent. The type of a component is not to be confused with the structural type of a module. In block diagrams, x_I is written $x{:}\,I$.

See Fig. 2 for the different port and connector types used in block diagrams.

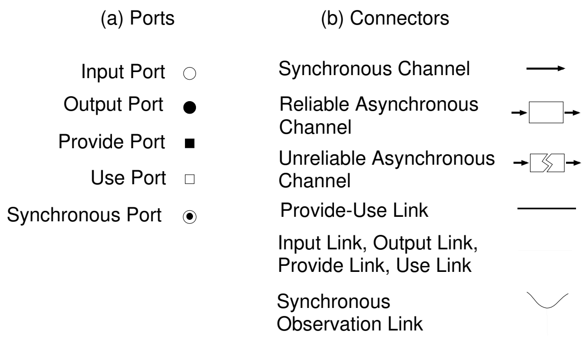

(a) Ports
Input Port
Output Port
Provide Port
Use Port
Synchronous Port
(b) Connectors
Synchronous Channel
Reliable Asynchronous Channel
Unreliable Asynchronous Channel
Provide-Use Link
Input Link, Output Link, Provide Link, Use Link
Synchronous Observation Link

Fig. 2: Ports and connectors.

2.3.3 Unions

A *union* (drawn as a rectangular box with cut corners) expresses bounded variability. It consists of a *finite* set of modules with identical interfaces. The modules which constitute a union are called its *members*. A union must have at least two distinct members.

Members of a union must have the same designation: they are either all interaction modules or all observation modules, giving rise to interaction and observation unions, respectively. Members cannot be environment modules. A member module inherits its interface from the union to which it belongs: $\mathcal{I}_M = \mathcal{I}_U$, for every member M of a union U.

Note the difference between the members of a union and the components of a composite. Whereas members are modules, components are instances. Components can be interconnected, but members (and modules) can not. However, instances of unions and their members can serve as components, and hence they can be interconnected within a composite.

If a composite M has a component x_U, where U is an interaction union, then it is possible to envision a *selection* operation which replaces that component with an instance of a particular member I of U. The resulting module is denoted by $(M \backslash x_U)[I]$. This operation will be discussed in detail in Section 3.

Two kinds of unions are possible: *functional unions* (marked by "**f**") and *variant unions* (marked by "**v**").

2.3.4 Functional Unions and Variant Unions

The members of a *functional union* represent interchangeable systems. Depending on the designation of the functional union, the members are expected either to behave similarly when used as components in larger systems (if the members are interaction modules) or to produce identical observations (if the members are observation modules).

By contrast, the members of a *variant union* represent related systems whose external functionalities or observable behaviors vary. For example, different types of users of an open system may be defined as members of a variant union.

Typically, a functional union contains an *abstract* member and a number of *concrete* members. The abstract member represents an idealized system, and the concrete members represent alternative implementations of the abstract member. The abstract member is called the *abstraction* of the union. Each functional union must declare one of its singular members as its abstraction. Hence, every functional union must possess at least one singular member. In block diagrams, the abstraction of a union can be identified by its bold frame.

2.3.5 Groups and Singularity

A *group*, defined recursively, is either (1) a union or (2) a composite with a component which is an instance of a group. Thus a composite group must possess a union submodule. A group represents a set of modules, and therefore, can be reduced to a union. In block diagrams, groups are indicated by the symbol "**g**".

A module which is not a group is *singular*. A singular module can not be reduced to a union. All primitive modules are by definition singular. All environment modules are also singular. A composite is singular if it does not have a union submodule.

A *functional group* is a group whose union submodules are all functional unions. Similarly, a *variant group* is one whose union submodules are all variant unions.

An *interaction group* is a group which is an interaction module, and an *observation group* is one which is an observation module. Functional groups play a special role in architecture-driven verification.

2.3.6 The Submodule Relation

The *submodule* relation captures both the *part-of* and the *member-of* relationships between modules. It is defined in terms of the *immediate-submodule* relation $\prec$. We say that M is an *immediate submodule* of N, written $M \prec N$, if and only if one of the following conditions is satisfied:

- N is a composite, M is an environment module, and M is a component of N;
- N is a composite, M is an interaction module, and N has a component x_M (of type M); or
- N is a union and M is a member of N.

The *submodule* relation $\prec^*$ is defined as the transitive closure of $\prec$. It underlies the module hierarchy of an architectural specification.

M is a called an *immediate subunion* (respectively, *immediate subgroup* and *immediate subcomposite*) of N if $M \prec N$ and N is a *union* (respectively, *group* and *composite*). The definition of *subunion* (respectively, *subcomposite* and *subgroup*) is obtained if $\prec^*$ is used instead of $\prec$.

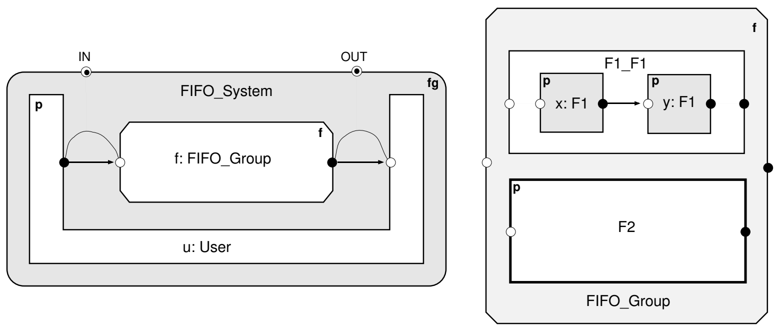


Fig. 3: Block diagrams for the FIFO buffers system.

A group can be defined in terms of the submodule relation. G is a group if either G is a union or it has a subunion. Equivalently, G is a group if either G is a union or it has an immediate subgroup.

2.4 A Small Example: Synchronous FIFO Buffers

Consider a system composed of a user module and a synchronous FIFO buffer with a maximum capacity of two slots. This system is illustrated in Fig. 3. The top-level module *FIFO_System* is defined as an observation group, where the component f is as an instance of a union *FIFO_Group*. Interactions between the components f and u can be observed through the observation ports *IN* and OUT.

The functional interaction union *FIFO_Group* has two members. One of these, the primitive module *F2* is defined as the abstraction of the union: it can be thought of as representing a centralized implementation of *FIFO_Group*. The other member *F1_F1* represents a distributed implementation of *FIFO_Group* in terms of the composition of two synchronous FIFO buffers each with a maximum capacity of one slot. Thus each of the components x and y of *F1_F1* is an instance of a primitive module *F1*.

Consequently, it is possible to generate two singular observation modules from the top level module *FIFO_System*. These singular modules are generated by the selections $(\mathit{FIFO\_System} \backslash f)[\mathit{F2}]$ and $(\mathit{FIFO\_System} \backslash f)[\mathit{F1\_F1}]$. Since *FIFO_Group* is defined as a functional union, these two selections are expected to produce identical observations.

3. Module Systems

Having established the basic structural model, it is time to provide the relevant formal definitions. Connections and connectors will be disregarded in this section, because they are not relevant for the discussion which follows.

3.1 Definition of a Module System

A module system $\mathcal{S}$ is a quadruple $\langle \mathcal{M}, \sigma, \delta, \Delta \rangle$ where

- $\mathcal{M}$ is a set of modules;
- σ is a partial function on $\mathcal{M}$ which may associate with $M \in \mathcal{M}$, a *structural type* $\sigma_M \in \{\mathbf{p}, \mathbf{c}, \mathbf{u}\}$;
- δ is a total function on $\mathcal{M}$ which associates with every $M \in \mathcal{M}$, a *designation* $\delta_M \in \{\mathbf{i}, \mathbf{o}, \mathbf{e}\}$; and
- Δ is a partial function on $\mathcal{M}$ which may associate with $M \in \mathcal{M}$, an *abstraction* $\Delta_M \in \mathcal{M}$.

The three structural types in the range of σ stand for **p**rimitive module, **c**omposite, and **u**nion, respectively. The three designations in the range of δ stand for **o**bservation module, **i**nteraction module, and **e**nvironment module, respectively.

The structural type of a module determines its structure:

- A primitive module $P \in \mathcal{M}$ is a pair $\langle \nu_P, \mathcal{I}_P \rangle$, where ν_P is the *name* of P and $\mathcal{I}_P$ is the interface of P. The name of a primitive module uniquely identifies it in $\mathcal{M}$.
- A composite $C \in \mathcal{M}$ is a quadruple $\langle \mathcal{I}_C, \mathcal{X}_C, \mathcal{C}_C, \tau_C \rangle$, where $\mathcal{I}_C$ is the interface of C, $\mathcal{X}_C$ is the set of *components* of C, $\mathcal{C}_C$ is the set of *connections* of C, and τ_C is a partial function from $\mathcal{X}_C$ to $\mathcal{M}$.
- A union $U \in \mathcal{M}$ is a pair $\langle \mathcal{I}_U, \mathcal{N}_U \rangle$, where $\mathcal{I}_U$ is the interface of $\mathcal{U}$ and $\mathcal{N}_U$ is a set of modules called the *members* of U. By abuse of notation, we will write $N \in U$ whenever $N \in \mathcal{N}_U$.

For composites, the partial function τ_C associates with every element x of $\mathcal{X}_C$, where x is *not* an environment module, a corresponding *type* $\tau_C(x)$ in $\mathcal{M}$. Note that the type of a component is a module, and the domain of τ_C gives the subset of those components which are instances of some interaction module. For $x \in \mathcal{X}_C$, the notation x_I is a shorthand for $\tau_C(x) = I$. For the purposes of this article, the details of connections are not relevant. The interested reader can find the formal definition in [22].

Definition 1. *A set of modules $\mathcal{M}$ is downwards closed under the relation $\prec$ if for every $N \in \mathcal{M}$, $M \prec N$ implies $M \in \mathcal{M}$. $\mathcal{M}^{\prec}$ is the smallest superset of $\mathcal{M}$ which is downwards closed under $\prec$.*

To be well defined, a module system must satisfy extra properties. For example, each functional union of $\mathcal{M}$ has at least one singular member; for every functional union U of $\mathcal{M}$, $\Delta_U \in U$ and U must be a singular module; etc. The formal definition of well-definedness for modules can be found in [22].

3.2 Contexts

A context $C_{x,I}$ is a composite module with a single *stub* component. A stub component $s_{x,I}$ is a placeholder for a component x whose interface must equal $\mathcal{I}_I$. An instance x_J (of an interaction module J) may be substituted for the stub $s_{x,I}$ in $C_{x,I}$ provided $\mathcal{I}_I = \mathcal{I}_J$. As such, contexts express unbounded variability. The result of the substitution is a composite, and is denoted by $C_{x,I}[J]$.

Mathematically, a context is simply a partial (unary) function from modules to modules. A context can be obtained from a composite C by making a particular component x_I of C a stub. A context obtained in this way is denoted by $C\backslash x_I$. If the stub x_I is substituted by module J, then one obtains the module $(C\backslash x_I)[J]$. Here, $C\backslash x_I$ is called the *context* of x_I in C. The result of the substitution, or the module, $(C\backslash x_I)[J]$ is called a *completion* of $C\backslash x_I$ and a *mutation* of C. The composite C is called the *enclosing module* of $C\backslash x_I$. Finally, the module J, which replaces the stub x_I, is called a *replacement* of $C\backslash x_I$. This construction is formalized in Definition 2:

DEFINITION 2. *Let C be a composite and $x_I \in \mathcal{C}_C$. Define*

$$(C\backslash x_I)[J] \stackrel{\text{def}}{=} \begin{cases} \langle \mathcal{I}_C, \mathcal{X}_C, \mathcal{C}_C, \tau_C \oplus \{x \mapsto J\}\rangle & \textit{if } \mathcal{I}_I = \mathcal{I}_J; \\ \textit{undefined} & \textit{otherwise,} \end{cases}$$

where

$$(\tau_C \oplus \{x \mapsto J\})(y) \stackrel{\text{def}}{=} \begin{cases} \tau_C(y) & \textit{if } x \neq y; \\ J & \textit{otherwise.} \end{cases}$$

Contexts may be *open* or *closed* depending on the designation of their enclosing modules. A context $C\backslash x$ is *closed* if C is an observation module, it is *open* if C is an interaction module.

By Definition 2, completion is commutative:

$$((C\backslash x)[I]\backslash y)[J] = ((C\backslash y)[J]\backslash x)[I] \text{ if } \; x \neq y.$$

In addition, $(C\backslash x_I)[I] = C$ for every context $C\backslash x_I$. Two contexts can be composed as follows:

DEFINITION 3. *Let C and D be two contexts such that the enclosing module of D has the same interface as the stub of C. Let I be an interaction module such that I has the same interface as (the type of) the stub of D. Then $C \circ D$, the composition of C and D, is a new context defined as*

$$(C \circ D)[I] \stackrel{\text{def}}{=} C[D[I]].$$

3.3 Selections

If the enclosing module of a context is a group, a special case arises. Let $G\backslash x$ be such a context (G is a group). Then its enclosing module G can recursively be specialized to a mutation G' by using replacements drawn from the members of the subunions (union submodules) of G. The resulting mutation G' is called a *selection* of G.

A selection can be defined recursively as follows:

DEFINITION 4. *Let G be a group.*

1. *If G is a union and $M \in G$ then M is a selection of G.*
2. *If G is a composite, $x_H \in \mathcal{C}_G$, and I is a selection of H, then $(G\backslash x_H)[I]$ is a selection of G. The designation of $(G\backslash x_H)[I]$ equals the designation of G and the structural type of $(G\backslash x_H)[I]$ is $\mathbf{c}$ (a composite).*
3. *If G'' is a selection of G' and G' is a selection of G then G'' is a selection of G.*
4. *Nothing else is a selection of G.*

The following shorthand notation is used for nested selections:

$$(G\backslash x_{H_1},\ldots,x_{H_n})[I_1,\ldots,I_n] \overset{\text{def}}{=} ((G\backslash x_{H_1},\ldots,x_{H_{n-1}})[I_1,\ldots,I_{n-1}]\backslash x_{H_n})[I_n]$$

For examples, consider the module system given in Fig. 4.

1. A and D are selections of U.
2. B and C are selections of V.
3. $(H\backslash u)[A]$ and $(H\backslash u)[D]$ are selections of H.
4. $(G\backslash h)[(H\backslash u)[A]]$ is a selection of G.
5. $(H\backslash u,v)[A,B] = ((H\backslash u)[A]\backslash v)[B] = ((H\backslash v)[B]\backslash u)[A] = (H\backslash v,u)[B,A]$ is a selection of $(H\backslash u)[A]$, $(H\backslash v)[B]$, and H.
6. $(G\backslash h)[(H\backslash u,v)[A,B]]$ is a selection of G.

DEFINITION 5. *Let $\mathcal{M}$ be a set of modules. $\mathcal{M}$ is closed under selection if for every group $G \in \mathcal{M}$, H is a selection of G implies $H \in \mathcal{M}$. $\mathcal{M}^*$, the closure of $\mathcal{M}$ under selection is the smallest superset of $\mathcal{M}$ closed under selection.*

Clearly, if $\mathcal{M}$ is downwards closed under $\prec$, then $\mathcal{M}^*$ is downwards closed under $\prec$.

Now Definition 5 can be extended to module systems.

DEFINITION 6. *Let $\mathcal{S} = \langle \mathcal{M}, \sigma, \delta, \Delta \rangle$ be a well defined module system. $\mathcal{S}^*$, the closure of $\mathcal{S}$ under selection is the module system $\langle \mathcal{M}^*, \sigma^*, \delta^*, \Delta \rangle$, where*

$$\sigma^*_M \overset{\text{def}}{=} \begin{cases} \sigma_M & \text{if } M \in \mathcal{M}; \\ \mathbf{c} & \text{otherwise} \end{cases} \qquad \delta^*_M \overset{\text{def}}{=} \begin{cases} \delta_M & \text{if } M \in \mathcal{M}; \\ \delta_G & \text{if } M \text{ is a selection of } G \in \mathcal{M}. \end{cases}$$

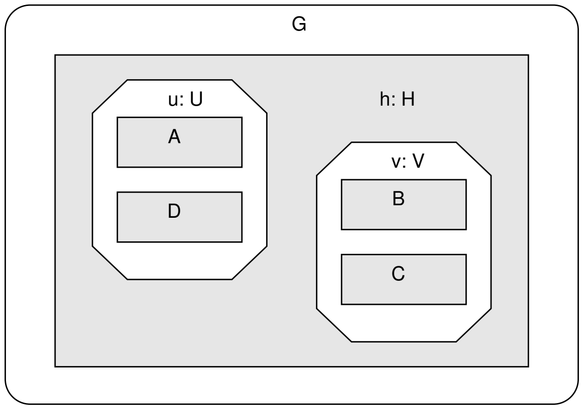


Fig. 4: A module system $\mathcal{M} = \{G, H, U, V, A, B, C, D\}$. Interfaces (ports) and connections are not shown.

In Definition 6, δ^* is well defined because the designation of a group is preserved by every selection of that group. $\mathcal{S}^*$ must be well defined since a selection cannot violate any of the properties a well defined module must satisfy, and a selection cannot be a union in a well defined module system.

4. Derivation and Expression of Correctness Requirements

4.1 Functional Classes

Consider a well defined module system $\langle \mathcal{M}, \sigma, \delta, \Delta \rangle$. Typically, $\mathcal{M}$ will contain some functional unions, and consequently, $\mathcal{M}^*$ will contain some functional groups. By definition, a functional group — if it is not a functional union itself — has at least one submodule which is a functional union.

A functional union represents a collection of interchangeable systems. A functional group involves functional unions, and therefore, it defines an equivalence class of (singular) modules. For the time being, it is sufficient to assume that such equivalence classes give rise to proof obligations which, given a behavioral model and a suitable mapping from modules to behaviors, can be discharged by checking the satisfaction of a behavioral relation between selected pairs of modules from the equivalence classes. This subject will be examined later.

Definition 7. *Let G be a group. A selection M of G is called a singular selection if M is a singular module. G^1 denotes the set of all singular selections of G.*

The equivalence class associated with a functional group F is called the *functional class* of F. It is given by F^1, the set of all singular selections of F. A singular selection has no selections of its own.

Functional groups that are observation modules are of special interest. These modules occupy nodes at the top layer of a module hierarchy (that is, they have no supermodules). As they represent closed systems, it is assumed that only the proof obligations of such modules can be discharged. The abbreviation *fog* refers to *functional observation group*. The term *functional observation class*, or *foc*, refers to the functional class of a *fog*.

4.2 The Abstraction of a Functional Group

The singular selections of a group can be interpreted as a union. This is possible because all selections of a group possess the same interface. Let U_G denote the union associated with a group G. Then U_G can simply be defined as $U_G \overset{\text{def}}{=} \langle \mathcal{I}_G, G^1 \rangle$. The designation of U_G equals that of G since the designation of a group is preserved by its selections.

If G is a variant group, U_G is automatically a variant union. If G is functional group, then for U_G to be interpreted as a functional union, an abstraction $\Delta_{U_G} \in G^1$ must be defined for U_G. In general, any module in G^1 can serve as the abstraction since all modules of G^1 are singular. However, one particular module in G^1 is a better candidate than the others. This particular selection of G — denoted by Δ^1_G — will be referred to as the *abstraction of* G. Intuitively, Δ^1_G is obtained from G by using only the abstractions of G's subgroups (group submodules) as replacements.

Definition 8. *Let $F \in \mathcal{M}^*$ be a functional group of a well defined module system $\langle \mathcal{M}, \sigma, \delta, \Delta \rangle$. For a composite group $H \in \mathcal{M}$, let $\mathcal{G}_H$ denote the set of all group components of H. That is,*

$$\mathcal{G}_H \overset{\text{def}}{=} \{x_G \in \mathcal{C}_H \mid G \in \mathcal{M} \textit{ is an interaction group}\}.$$

Define $\Delta^1_F \in F^1$, the abstraction of F, as the following singular selection of F:

1. *If F is a union, then $\Delta^1_F \overset{\text{def}}{=} \Delta_F$.*
2. *If F is a composite with $\mathcal{G}_F = \{x_{G_1}, \ldots, x_{G_n}\}$, then*

$$\Delta^1_F \overset{\text{def}}{=} (F \backslash x_{G_1}, \ldots, x_{G_n})[\Delta^1_{G_1}, \ldots, \Delta^1_{G_n}].$$

4.3 Obligations

The functional observation groups (*fog*s) of a module system give rise to proof obligations. Again, let $\mathcal{S} = \langle \mathcal{M}, \sigma, \delta, \Delta \rangle$ be a well defined module system.

Definition 9. *A proof obligation, or simply an obligation o of $\mathcal{S}$ is a two element set $\{M, N\}$ where:*

1. *M and N are distinct singular modules in $\mathcal{M}^*$, and*
2. *there exists a fog $F \in \mathcal{M}^*$ such that both M and N belong to F^1, the foc of F.*

Obligation satisfaction is defined with respect to a behavioral model $\langle \mathcal{B}, \equiv \rangle$ and a mapping β from the singular modules of $\mathcal{M}^*$ to $\mathcal{B}$. Here $\mathcal{B}$ is a set of behavior descriptions and $\equiv$ is an equivalence relation over $\mathcal{B}$.

DEFINITION 10. *An obligation $\{M, N\}$ of a module system is satisfied for the model $\langle \mathcal{B}, \equiv \rangle$ under the mapping β if $\beta(M) \equiv \beta(N)$. A set of obligations $\mathcal{O}$ is satisfied for $\langle \mathcal{B}, \equiv \rangle$ under β if every obligation $o \in \mathcal{O}$ is satisfied for $\langle \mathcal{B}, \equiv \rangle$ under β.*

DEFINITION 11. *Let $F \in \mathcal{M}^*$ be a fog. Define*

$$\mathcal{O}_F \overset{\text{def}}{=} \{\{M, N\} \mid M, N \in F^1\}.$$

$\mathcal{O}_F$ is called the obligations of F.

It is possible to compose two obligations provided they involve a common module:

DEFINITION 12. *Let o and o' be two obligations such that $o \neq o'$ and $o \cap o' \neq \emptyset$. The composition $o \circ o'$ of o and o' is defined as:*

$$o \circ o' \overset{\text{def}}{=} (o \cup o') \setminus (o \cap o').$$

A set of obligations $\mathcal{O}$ is closed under composition if $o, o' \in \mathcal{O}$ implies $o \circ o' \in \mathcal{O}$ for every o, o' such that $o \neq o'$ and $o \cap o' \neq \emptyset$.

The two conditions of obligation composition guarantee that $o \circ o'$ is an obligation whenever o and o' are obligations. If $o = \{L, M\}$ and $o' = \{M, N\}$ where $L \neq N$, then $o \circ o' = \{L, N\}$. Note $\mathcal{O}_F$ is closed under composition for every *fog* F.

4.4 Design Hypotheses

Hypotheses specify dependencies among obligations in an *assume-guarantee* style. A *hypothesis* h is written as

$$h : o_1, \ldots, o_n \vdash o$$

The obligations o_i are called the *assumptions* of h and the obligation o is called the *guarantee* of h; $\alpha(h)$ denotes the assumptions and $\gamma(h)$ denotes the guarantee.

A hypothesis h is said to be *valid* if the guarantee $\gamma(h)$ is satisfied whenever each of the obligations in the assumptions $\alpha(h)$ is individually satisfied. Thus a valid hypothesis allows the decomposition of its guarantee into its assumptions.

Two hypotheses can be composed as follows:

DEFINITION 13. *Let h and h' be two hypotheses. The composition of h and h' is the hypothesis $h \circ h' : \alpha(h), \alpha(h') \setminus \gamma(h) \vdash \gamma(h')$.*

Validity is preserved by composition: if h and h' are valid hypotheses, then so is their composition $h \circ h'$.

Validity is extended to a set of hypotheses in the obvious manner: $\mathcal{H}$ is *valid* if all of its hypotheses are valid.

4.4.1 Implicit Hypotheses

Within a well defined module system, every *fog* F is associated with a set $\mathcal{H}_F$ of valid hypotheses. The hypotheses of $\mathcal{H}_F$ are derived from $\mathcal{O}_F$, the obligations of F. Keep in mind that F^1, the *foc* of F, represents an equivalence class. The set $\mathcal{H}_F$ is referred to as the *implicit hypotheses* of F.

Let $o, o' \in \mathcal{O}_F$ be two distinct obligations such that $o \cap o' \neq \emptyset$. Define $h_{o,o'}$ as

$$h_{o,o'} : o, o' \vdash o \circ o'.$$

The hypothesis $h_{o,o'}$ is valid because satisfaction for obligations is defined with respect to an equivalence relation (see Definition 10). This is also the reason F^1 can be viewed as an equivalence class. Hence, if $o = \{M_1, M_2\}$, $o' = \{M_2, M_3\}$, and if both o and o' are satisfied, then so is $\gamma(h_{o,o'}) = \{M_1, M_3\}$. Note that by definition, $\{M_1, M_2, M_3\} \subseteq F^1$ and $\gamma(h_{o,o'}) \in \mathcal{O}_F$. The validity of $h_{o,o'}$ implies that it is sufficient to discharge o and o' to verify the satisfaction of $\gamma(h_{o,o'})$.

The implicit hypotheses of F can now be defined as follows:

DEFINITION 14. *Let F be a foc.*

$$\mathcal{H}_F \stackrel{\text{def}}{=} \{h_{o,o'} \mid (o \neq o') \wedge (o, o' \in \mathcal{O}_F) \wedge (o \cap o' \neq \emptyset)\}.$$

$\mathcal{H}_F$ is valid since the $h_{o,o'}$ are valid. In addition, by Definition 11, $\mathcal{H}_F$ is closed under composition. The validity of $\mathcal{H}_F$ makes it possible to discharge a relatively small subset of $\mathcal{O}_F$ to fulfill all of F's obligations.

As an example refer to the module system depicted in Fig. 3. Here *FIFO_System* is a *fog*. Its *foc* *FIFO_System*^{1} consists of the singular modules $(\mathit{FIFO\_System} \backslash f)[F2]$ and $(\mathit{FIFO\_System} \backslash f)[F1\_F1]$. Thus, $\mathcal{O}_{\mathit{FIFO\_System}}$ contains a single obligation

$$\{(\mathit{FIFO\_System} \backslash f)[F2], (\mathit{FIFO\_System} \backslash f)[F1\_F1]\}.$$

Consequently, *FIFO_System* has no implicit hypotheses: $\mathcal{H}_{\mathit{FIFO\_System}} = \emptyset$.

4.4.2 Approximations and Explicit Hypotheses

It is possible to extend a module system with additional hypotheses to reduce further the number of obligations that must be discharged. The additional hypotheses can be represented concisely as approximations. An *approximation* is like an abstraction, but it applies to contexts rather than modules. It allows the decomposition of complex obligations into simpler obligations that are easier to check for satisfaction. Similar to abstractions, approximations are defined with respect to a functional group. Let

- closed contexts C and $D_1, \ldots, D_n$ whose stubs are of type I; and
- a functional interaction union U such that $\mathcal{I}_U = \mathcal{I}_I$.

Then $D_1, \ldots, D_n$ is said to *approximate C with respect to U* if for every $M, M' \in U^1$, there exists a hypothesis

$$\{D_1[M], D_1[M']\}, \ldots, \{D_n[M], D_n[M']\} \vdash \{C[M], C[M']\}.$$

This approximation is conveniently denoted by the tuple $\langle D_1, \ldots, D_n, C, U \rangle$. For a set of modules $\mathcal{M}$, the approximation is said to be *over* $\mathcal{M}$ if (1) $U \in \mathcal{M}$, (2) the enclosing modules of the contexts D_i are in $\mathcal{M}^*$ or each D_i is composed of contexts whose enclosing modules are in $\mathcal{M}^*$, and (3) the enclosing module of C is in $\mathcal{M}$.

The approximation $\langle D_1, \ldots, D_n, C, U \rangle$ asserts the ability of the contexts D_i to simulate the context C (a) in terms of the obligations associated with the completions of C, and (b) under replacements from the functional class of U.

An approximation is *valid* if all of its hypotheses are valid. The problem of obtaining valid approximations has been addressed elsewhere; for examples, see [38,15,12]. Validity of an approximation can be either guaranteed by construction or checked explicitly.

4.5 Designs

A design extends a module system with a set of approximations, a behavioral model, and a mapping from modules to behaviors. The approximations specify how the obligations of the module system can be decomposed and the behavioral model with the mapping defines the satisfaction criterion for the obligations.

A *design* $\mathcal{D}$ is a quadruple $\langle \mathcal{S}, \mathcal{A}, \langle \mathcal{B}, \equiv \rangle, \beta \rangle$, where

- $\mathcal{S} = \langle \mathcal{M}, \sigma, \delta, \Delta \rangle$ is a well defined module system;
- $\mathcal{A}$ is a set of approximations over $\mathcal{M}$;
- $\langle \mathcal{B}, \equiv \rangle$ is a behavioral model with $\mathcal{B}$ being a set of behavior descriptions and $\equiv$ an equivalence relation on $\mathcal{B}$; and
- $\beta \colon \mathcal{M}^1 \mapsto \mathcal{B}$ is a mapping from the singular modules of $\mathcal{M}^*$ to the set $\mathcal{B}$.

Satisfaction for obligations is defined for the behavioral model $\langle \mathcal{B}, \equiv \rangle$ under the mapping β (see Definition 10). Assume that $\mathcal{B}$ has a suitable notion of composition and that for a composite M, $\beta(M)$ is a function of the structure of M as well as the behaviors of M's submodules.

To give rise to a nonempty set of obligations, $\mathcal{M}^*$ must contain at least one *fog*. The following definitions will be needed in the sections to come:

- The *fogs of the design* $\mathcal{D}$ are given by the *fog*s of $\mathcal{M}^*$, and will be denoted by $\mathcal{F}_\mathcal{D}$.
- The *obligations of* $\mathcal{D}$ are given by $\bigcup_{F \in \mathcal{F}_\mathcal{D}} \mathcal{O}_F$, and will be denoted by $\mathcal{O}_\mathcal{D}$.
- The *implicit hypotheses of* $\mathcal{D}$ are given by $\bigcup_{F \in \mathcal{F}_\mathcal{D}} \mathcal{H}_F$, and will be denoted by $\mathcal{H}^i_\mathcal{D}$.
- The *explicit hypotheses of* $\mathcal{D}$ are given by the union of the sets of hypotheses of the approximations in $\mathcal{A}$, and will be denoted by $\mathcal{H}^e_\mathcal{D}$.
- The *hypotheses of* $\mathcal{D}$ are given by $\mathcal{H}^i_\mathcal{D} \cup \mathcal{H}^e_\mathcal{D}$, and will be denoted by $\mathcal{H}_\mathcal{D}$.

5. Obligation Decomposition

An obligation $\{M, N\}$ may be infeasible to discharge because $\beta(M)$ or $\beta(N)$ is infeasible to compute or too large for a $\equiv$-equivalence test. In this case, the obligation needs to be decomposed into obligations that are easier to discharge. Since obligations are derived from the design, they can be decomposed by decomposing the design itself.

Design decomposition in this sense involves a transformation of the underlying module system together with the proper extension of its set of approximations. The result is a syntactically legal decomposition if the new design has a larger set of obligations than those of the original design. In addition, there must exist a subset of the original obligations such that each obligation in this subset is the guarantee of some hypothesis of the new design.

Two kinds of decompositions are introduced: (1) *horizontal decomposition*, which directly exploits the property underlying the obligations to be decomposed; and (2) *vertical decomposition*, which exploits the hierarchical structure of a module to decompose an obligation involving that module. In what follows, assume $\mathcal{D} = \langle \mathcal{M}, \mathcal{A}, \langle \mathcal{B}, \equiv \rangle, \beta \rangle$.

5.1 Horizontal Decomposition

Let

- $H \in \mathcal{M}$ be a *functional interaction group*; and
- $D[H] \in \mathcal{M}$ be a *fog*, where D is a closed context.

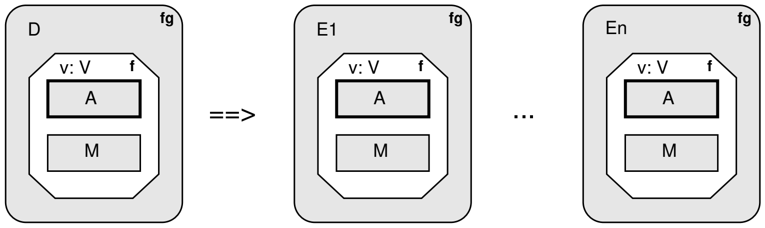


Fig. 5: Horizontal decomposition: $E_1, \ldots, E_n$ approximates D with respect to V.

Assume the obligations of $D[H]$ are to be decomposed. Then $\mathcal{D}$ can be decomposed into $\mathcal{D}'$ by introducing n new *fog*s $E_1[H], \ldots, E_n[H]$ such that $E_1, \ldots, E_n$ *approximates* D *with respect to* H. The resulting design $\mathcal{D}'$ is such that

- $\mathcal{M}' = \mathcal{M} \cup \{E_1[H], \ldots, E_n[H]\}^{\prec}$
- $\mathcal{A}' = \mathcal{A} \cup \{\langle E_1, \ldots, E_n, D, H\rangle\}$.

Horizontal decomposition is illustrated in Fig. 5.

5.2 Vertical Decomposition

Let

- $F \in \mathcal{M}$ be a *fog*;
- $I \in \mathcal{M}$ be a singular interaction module; and
- $C[I] \in F^1$, where C is a closed context.

Assume the obligations of F which involve $C[I]$ are to be decomposed. Then $\mathcal{D}$ can be decomposed into $\mathcal{D}'$ as follows:

1. Introduce a new functional union V such that $I \in V$. The abstraction of V must be different from I ($\Delta_V \neq I$). For the decomposition to pay off, the abstraction Δ_V should be considerably simpler than I.
2. Replace F by F' such that

$$(F')^* = F^* \cup \{C[V]\}^*.$$

   Therefore, the submodule I of F is replaced by the union V.
3. Define a new closed context D for V such that D *approximates* C *with respect to* V. Include the *fog* $D[V]$ in $\mathcal{M}$. Typically D is simpler than C.

The resulting design $\mathcal{D}'$ is such that

- $\mathcal{M}' = (\mathcal{M} \setminus \{F\}) \cup \{F', D[V]\}^{\prec}$
- $\mathcal{A}' = \mathcal{A} \cup \{\langle D, C, V\rangle\}$.

Note that $\{F'\}^{\prec}$ includes V, I, and Δ_V.

Vertical decomposition is illustrated in Fig. 6.

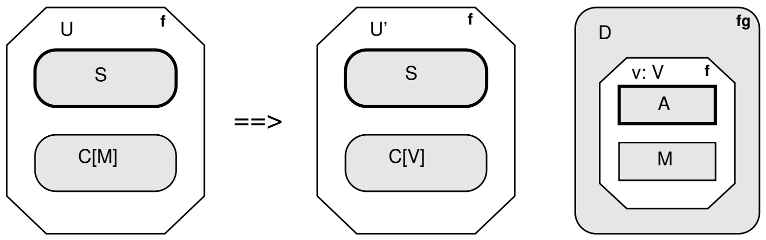
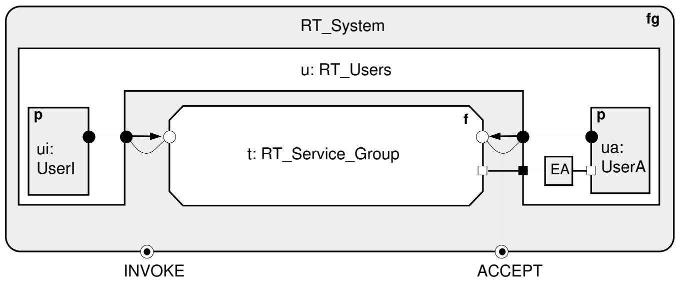


Fig. 6: Vertical decomposition: D approximates C with respect to V.

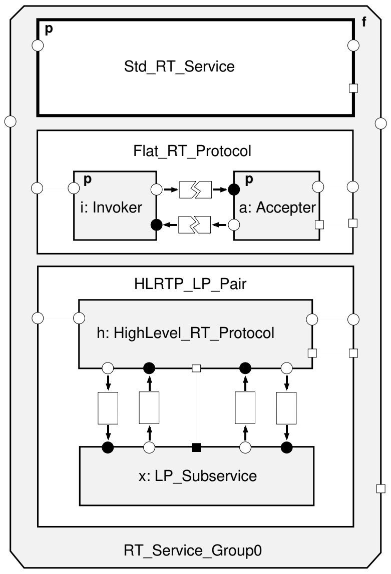
5.3 Case Study: Decomposition of A Protocol

As an example, consider the architecture of a system consisting of a protocol module and two users [24,23]. The functionalities of the modules involved are discussed only informally to provide a rationale for the example; no particular behavioral model is assumed. The top level system is given by the *fog RT_System*:

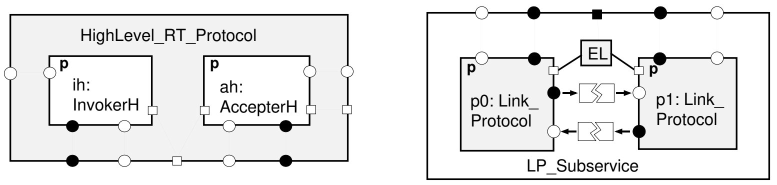


RT_System consists of two components: a component u of type *RT_Users* and a component t of type *RT_Service_Group*. The component t (module *RT_Service_Group*) represents the protocol whose purpose is to provide remote method invocation service to end users over unreliable and semi-reliable connections. The component u (module *RT_Users*) is in turn composed of an invoking user *ui*, an accepting user *ua*, and an environment module *EA*.

An initial breakdown of the functional union *RT_Service_Group* is given by the following module (named *RT_Service_Group0* in the block diagram):



The primitive member *Std_RT_Service* of this functional union represents the abstract service to be provided to the two end users, and is declared as the abstraction of the functional union. The member *Flat_RT_Protocol* provides the required service over an unreliable connection in terms of a single-layer protocol. The third member *HLRTP_LP_Pair* provides the same service over a semi-reliable connection by means of a higher-level protocol (module *HighLevel_RT_Protocol*), which in turn relies on a data link module (*LP_Subservice*) for data transmission. These two modules are in turn expanded as follows:



Note that in *Flat_RT_Protocol* the components i and a communicate directly over unreliable connections, whereas in *HighLevel_RT_Protocol* the corresponding components ih and ah do not.

The *fog RT_System* has three singular selections, namely:

$R'_1 = (RT\_System\backslash t)[Std\_RT\_Service]$,

$R'_2 = (RT\_System\backslash t)[Flat\_RT\_Protocol]$, and

$R'_3 = (RT\_System\backslash t)[HLRTP\_LP\_Pair]$.

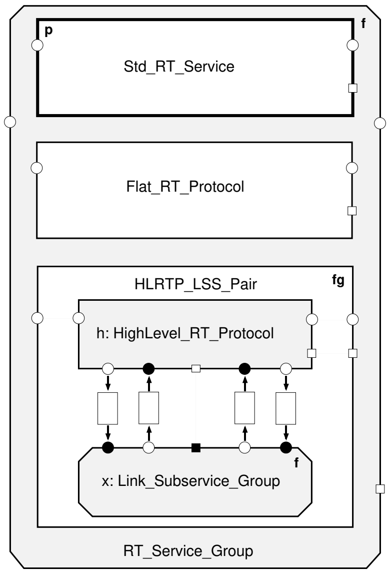
Thus, the underlying *foc* RT_System^1 equals $\{R'_1, R'_2, R'_3\}$. This *foc* gives rise to three initial obligations:

$$o'_1 = \{R'_1, R'_2\}, o'_2 = \{R'_1, R'_3\}, o'_3 = \{R'_2, R'_3\}.$$

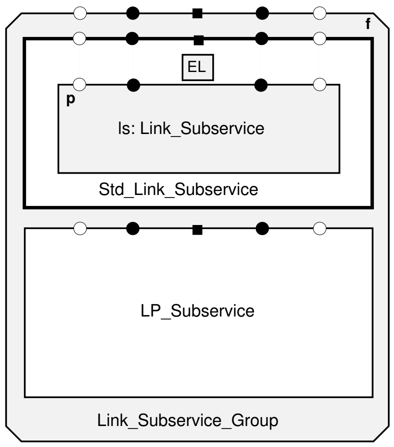
Note that obligation o'_3 is already covered by obligations o'_1 and o'_2 due to the implicit hypothesis $o'_1, o'_2 \vdash o'_3$, and therefore, o'_3 is redundant (optional).

5.3.1 Vertical Decomposition

Suppose obligation o'_2 is infeasible to discharge because $\beta(R'_3)$ is infeasible to compute. To apply vertical decomposition, first *RT_Service_Group* is redefined as follows:



Here the functional group *HLRTP_LSS_Pair* replaces the module *HLRTP_LP_Pair*. *HLRTP_LSS_Pair* is similar to *HLRTP_LP_Pair* except that the component x is an instance of *Link_Subservice_Group*, a functional union:



This functional union includes an abstract member *Std_Link_Subservice*, as well as the more complex module *LP_Subservice*. Whereas the first module *Std_Link_Subservice* represents an abstract data link service, the second module *LP_Subservice* represents a concrete implementation of this service in terms of two peer protocol modules communicating over an unreliable connection.

Now *RT_System* has the following singular selections:

$$R_1 = (\mathit{RT\_System}\backslash t)[\mathit{Std\_RT\_Service}],$$

$$R_2 = (\mathit{RT\_System}\backslash t)[\mathit{Flat\_RT\_Protocol}],$$

$$R_3 = (\mathit{RT\_System}\backslash t)[(\mathit{HLRTP\_LSS\_Pair}\backslash x)[\mathit{LP\_Subservice}]],$$

$$R_4 = (\mathit{RT\_System}\backslash t)[(\mathit{HLRTP\_LSS\_Pair}\backslash x)[\mathit{Std\_Link\_Subservice}]].$$

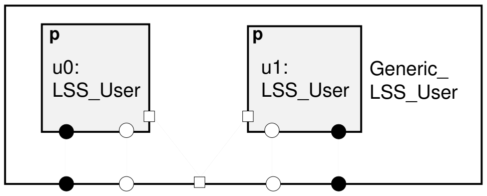
Here, R_1 and R_2 are identical to R'_1 and R'_2, respectively, and R_3 is equivalent to R'_3. The new set of obligations are:

$$o_1 = \{R_1, R_2\}, o_2 = \{R_1, R_3\}, o_3 = \{R_2, R_3\},$$

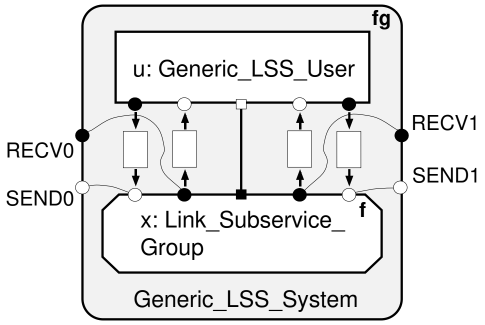
$$o_4 = \{R_3, R_4\}, o_5 = \{R_1, R_4\}, o_6 = \{R_2, R_4\}.$$

A similar correspondence exists between the new and the old obligations with identical indices.

To complete the vertical decomposition, an appropriate closed context for *Link_Subservice_Group* and a corresponding approximation must be introduced. The context involves the symmetric, general purpose user module *Generic_LSS_User*:



The resulting *fog* is the following one:



The approximation associated with vertical decomposition is:

Generic_LSS_System$\backslash x$ approximates
$(\mathit{RT\_System}\backslash t) \circ (\mathit{HLRTP\_LSS\_Pair}\backslash x)$
with respect to *Link_Subservice_Group*.

Let us denote this approximation by $\mathbf{V}$.

The *fog* *Generic_LSS_System* has two singular selections:

$$G_1 = (\mathit{Generic\_LSS\_System} \backslash x)[\mathit{Std\_Link\_Subservice}],$$

$$G_2 = (\mathit{Generic\_LSS\_System} \backslash x)[\mathit{LP\_Subservice}].$$

Therefore, this *fog* gives rise to a single obligation

$$o_7 = \{G_1, G_2\}.$$

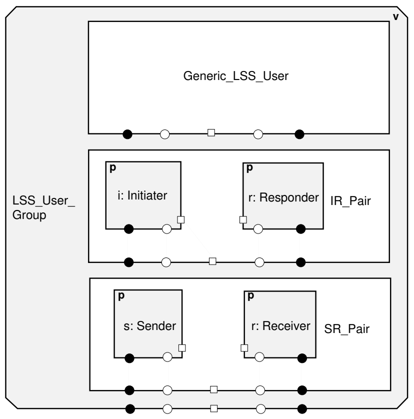
Finally, the explicit hypothesis associated with the approximation $\mathbf{V}$ is:

$$h_{\mathbf{V}} : o_7 \vdash o_4.$$

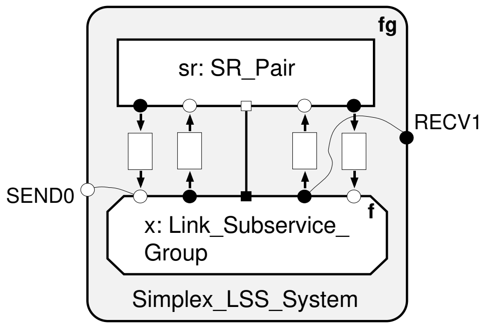
5.3.2 Horizontal Decomposition

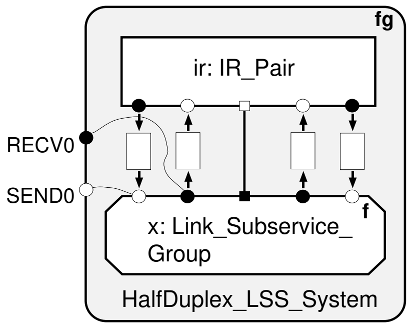
Now assume the obligation o_5 is infeasible to discharge. The following illustrates how horizontal decomposition can be applied to remedy the situation.

The *fog* *Generic_LSS_System* assumes the user module *Generic_LSS_User* engages in full-duplex communication with *Link_Subservice_Group*. The idea is to break down *Generic_LSS_User* into simpler user modules which engage in more restricted patterns of interaction with *Link_Subservice_Group*. The variant union *LSS_User_Group* is introduced for this purpose:



The member *SR_Pair* defines two asymmetric users in sender/receiver-type, simplex communication. The member *IR_Pair* defines two asymmetric users in half-duplex, initiator/responder-type communication. A *fog* is defined for each of these new, more restricted types of users: *Simplex_LSS_System* for *SR_Pair* and *HalfDuplex_LSS_System* for *IR_Pair*:





Note the differences among the interfaces of the three *fog*s shown in the figure. The following approximation completes the horizontal decomposition:

> *Simplex_LSS_System*$\backslash x$, *HalfDuplex_LSS_System*$\backslash x$
> approximates *Generic_LSS_System*$\backslash x$
> with respect to *Link_Subservice_Group*.

Denote this second approximation by **H**. Intuitively, **H** asserts that it would be sufficient to test *Link_Subservice_Group* separately in simplex and in half-duplex operation to infer that it behaves correctly in full-duplex operation. (This assertion is based on the symmetric nature of the service provided by *Link_Subservice_Group* to its users.)

The *fog Simplex_LSS_System* has two singular selections, namely

$$S_1 = (\mathit{Simplex\_LSS\_System}\backslash x)[\mathit{Std\_Link\_Subservice}],$$

$$S_2 = (\mathit{Simplex\_LSS\_System}\backslash x)[\mathit{LP\_Subservice}].$$

and a single obligation

$$o_8 = \{S_1, S_2\}.$$

Similarly, the *fog HalfDuplex_LSS_System* has two singular selections, namely

$$H_1 = (\mathit{HalfDuplex\_LSS\_System}\backslash x)[\mathit{Std\_Link\_Subservice}],$$

$$H_2 = (\mathit{HalfDuplex\_LSS\_System}\backslash x)[\mathit{LP\_Subservice}],$$

and a single obligation

$$o_9 = \{H_1, H_2\}.$$

Therefore, the explicit hypothesis associated with the approximation **H** is

$$h_{\mathbf{H}} : o_8, o_9 \vdash o_7.$$

5.4 Obligation Selection

The hypotheses of $\mathcal{D}$ can be taken into account to reduce the number of obligations to be discharged. Provided all the explicit hypotheses of $\mathcal{D}$ are valid, it should be sufficient to discharge only a relatively small subset of $\mathcal{O}_\mathcal{D}$ to achieve full coverage. For a single *fog*, the size of this subset is linear in the size of the associated *foc*. If suitable approximations are defined, it can even be linear in the number of the subgroups involved.

A set of obligations is called *optimal* under a set of hypotheses if no obligation can be removed from it without affecting the coverage of the set. It is *sufficient* with respect to a larger set if it provides full coverage for the latter under a given set of hypotheses [22]. The aim is to obtain a subset of $\mathcal{O}_{\mathcal{D}}$ that is both optimal under the hypotheses of $\mathcal{D}$ and sufficient with respect to $\mathcal{O}_{\mathcal{D}}$ under the hypotheses of $\mathcal{D}$. The desired subset can be constructed in many different ways. A reasonable starting point is the so-called *principal obligations* which involve the abstractions of the *fog*s of $\mathcal{D}$. The criteria used in obligation selection is addressed in detail in [22]. Here we only provide an example.

5.5 Case Study Revisited

Refer to the case study that was presented in Section 5.3. This case study involved a module system with four *fog*s (*RT_System*, *Generic_LSS_System*, *Simplex_LSS_System*, and *HalfDuplex_LSS_System*) and two approximations:

$$\begin{aligned}\mathbf{V} &= \langle \mathit{Generic\_LSS\_System}\backslash x, (\mathit{RT\_System}\backslash t) \circ (\mathit{HLRTP\_LSS\_Pair}\backslash x), \\ &\qquad \mathit{Link\_Subservice\_Group}\rangle,\end{aligned}$$

$$\begin{aligned}\mathbf{H} &= \langle \mathit{Simplex\_LSS\_System}\backslash x, \mathit{HalfDuplex\_LSS\_System}\backslash x, \\ &\qquad \mathit{Generic\_LSS\_System}\backslash x, \mathit{Link\_Subservice\_Group}\rangle.\end{aligned}$$

The set of obligations of the resulting design $\mathcal{D}$ is given by

$$\mathcal{O}_{\mathcal{D}} = \{o_1, \ldots, o_9\}$$

which is the union of the obligations of the four *fog*s. For the definitions of the o_i, refer again to Section 5.3.

The first step is to obtain the set of principal obligations $\mathcal{P}_{\mathcal{D}}$ (a principal obligation is one which involves an abstraction of some *fog*) which is sufficient with respect to $\mathcal{O}_{\mathcal{D}}$ under $\mathcal{H}_{\mathcal{D}}$:

$$\mathcal{P}_{\mathcal{D}} = \{o_1, o_2, o_5, o_7, o_8, o_9\}.$$

$\mathcal{P}_{\mathcal{D}}$ excludes obligations o_3, o_4, and o_6. For instance, o_3 is excluded due to the implicit hypothesis $o_1, o_2 \vdash o_3$. (Note that $o_3 = o_1 \circ o_2$.)

Now $\mathcal{P}_{\mathcal{D}}$ can be reduced by taking into account the explicit hypotheses of the two approximations: $\mathcal{H}^e_{\mathcal{D}} = \{h_{\mathbf{V}}, h_{\mathbf{H}}\}$ where $h_{\mathbf{V}} : o_7 \vdash o_4$ and $h_{\mathbf{H}} : o_8, o_9 \vdash o_7$.

Let $\mathcal{O} = \mathcal{O}_{\mathcal{D}}$.

1. Eliminate obligation o_2 from $\mathcal{O}$. Obligation o_2 can be removed because there exists an implicit hypothesis $h : o_4, o_5 \vdash o_2$ with $h_{\mathbf{V}} \circ h = o_7, o_5 \vdash o_2$. Therefore, o_2 is optional (redundant) in $\mathcal{O}$ under $\mathcal{H}_{\mathcal{D}}$. This step leaves
$$\mathcal{O} = \{o_1, o_5, o_7, o_8, o_9\}.$$
2. Eliminate obligation o_7 from $\mathcal{O}$. Obligation o_7 can be removed due to $h_{\mathbf{H}}$ (o_7 is optional in $\mathcal{O}$ under $\mathcal{H}_{\mathcal{D}}$ at the end of Step 1.) This step leaves
$$\mathcal{O} = \{o_1, o_5, o_8, o_9\},$$
which is both sufficient and optimal. An obligation can not be removed from this final set without violating its sufficiency with respect to $\mathcal{O}_{\mathcal{D}}$ under $\mathcal{H}_{\mathcal{D}}$.

6. Specification and Verification of Architectures in SPIN

Spin is a powerful modeling and analysis tool for concurrent systems [32,33]. It is based on the specification language Promela in which a system is described as a network of extended state machines (processes) communicating through FIFO channels.

Spin supports, among other things, model checking and random simulation. A front-end tool can translate correctness properties expressed in linear temporal logic to ω-automata. Then the properties can be verified by computation of the state space of a given Promela specification on the fly.

An experimental extension to Spin, the SpinE tool also supports another form of model checking: testing a given behavioral relation between two Promela specifications. This tool is similar to Spin except that it accepts two Promela specifications as input and generates a customized relation checker. The relation checker tests whether a particular behavioral relation holds true between the two input specifications. The flow diagram of the SpinE tool is given in Fig. 7. Currently SpinE is a prototype which supports only a particular class of behavioral relations. This class, while it includes such well known relations as trace equivalence [16] and testing equivalence [9,30], excludes some stronger relations such as observation (weak bisimulation) equivalence [46] which cannot be traced [8]. (Testing equivalence is not yet implemented in SpinE.) For more details about the SpinE tool, the reader is referred to [21,24,23].

To support relation checking it was necessary to extend the syntax of Promela. The new syntax allows a channel to be declared as an *external* channel and given a unique *external* name. External channels define the relevant observable behavior of a Promela specification so that this behavior can be compared with the observable behavior of another Promela specification. If the same name is used for two different channels in two Promela specifications, then the relation checker considers *send* and *receive* operations on these channels as matchable external transitions.

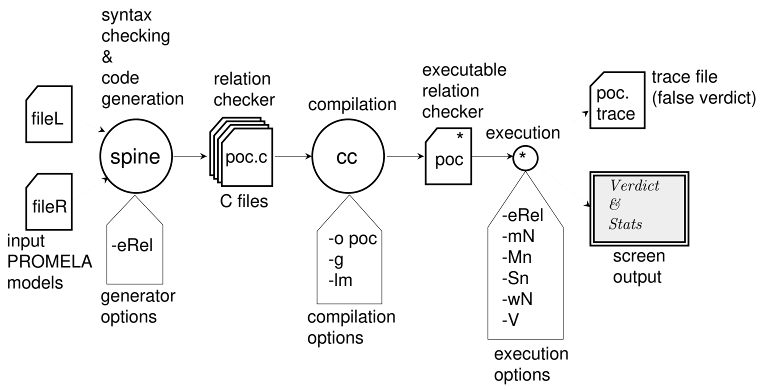


Fig. 7: Flow diagram of the SPINe verification system.

6.1 Representation of Architectures in PROMELA

Assume that PROMELA is used as the behavioral model for system architectures. Therefore the mapping β referred to in Section 4.5 associates a PROMELA specification $\beta(M)$ with each module M of a module system $\mathcal{M}$. Each PROMELA specification is stored in a separate file with the same name as the module. An appropriate suffix and prefix are added to the file name, depending on the designation and the structural type of the module. See [24,23,22] for more details.

If M is a primitive module, the behavior of M is given by a PROMELA process type. If M is a composite, the behavior of M is specified in terms of the behaviors of M's submodules. The behavior of an environment module consists of variable declarations and access macros.

The behavior of the primitive module *Std_RT_Service* is given in Fig. 8. First a process type is defined for the primitive module. This definition is followed by a `#define` statement. The macro defined by the `#define` statement has the same name as the primitive module. Its right hand side simply instantiates the process type and executes it. Each input and output interface port of the module is represented by a corresponding parameter of this macro. Process types are only defined for primitive modules.

In Fig. 9, the PROMELA specification of the *fog RT_System* is shown. Note that some of the channels are declared as external channels. In an interaction composite, the `init` statement is replaced by a `#define` statement. Here `InitEA` is a macro defined in the file `EA.env` which specifies an environment module *EA*. The macro initializes the shared variables declared in this file. The include file `Common.env` contains message type definitions and other macros accessible by all modules. For example, it defines generic send and receive operations to model nondeterministic message loss for unreliable channels.

Unions are represented in PROMELA in a straight forward manner as a

```
proctype _Std_RT_Service(chan fromUI, fromUA) {
restart:
  fromUI?RTinvoke -> if
                       :: RTServiceAvailable -> StartRT
                       :: RTServiceNotAvailable -> goto restart
                     fi;
                     do
                       :: fromUA?RTcomplete -> goto restart
                       :: fromUA?Pull
                       :: fromUA?Push
                       :: UserAError -> goto restart
                       :: RTServiceFailure -> goto restart
                     od }
#define Std_RT_Service(spI, spA)
  run _Std_RT_Service(spI, spA)
```

Fig. 8: The file `Std_RT_Service.int`.

```
#include "Common.env"
#include "RT_Users.int"
#include "%RT_Service_Group.int"
chan UItoRT (extern INVOKE) = [0] of {byte};
chan UAtoRT (extern ACCEPT) = [0] of {byte};
init {
  atomic{InitEA;
         RT_Users(UItoRT, UAtoIRT);
         RT_Service_Group(UItoRT, UAtoRT)} }
```

Fig. 9: The file `%RT_System.obs`.

series of "`#if ... #endif`" blocks that implement a logical case statement.

6.2 Discharging Obligations in SPIN

Considering β maps every module to a corresponding PROMELA specification, an obligation $\{M, N\}$ is discharged using the SPINe tool by checking $\beta(M) \equiv \beta(N)$, where $\equiv$ is a behavioral relation on PROMELA specifications. Trace equivalence was used as the correctness criterion in the case study of Section 5.3 and in other small examples tried. Thus $\equiv$ was taken to be trace equivalence.

The choice of the equivalence relation depends on the particular class of behavioral properties that equivalent specifications must preserve. That trace equivalence can preserve only safety properties is a well known fact. To reason about liveness properties, a stronger relation such as testing equivalence should be used.

Obligations often involve selections. A selection can be represented easily in PROMELA as a series of `#include` and `#define` statements. As an

```
#include "Mods.env"
#define nLink_Subservice_Group nStd_Link_Subservice
#define nRT_Service_Group nHLRTP_LSS_Pair
#include "%RT_System.obs"
```

Fig. 10: Specification of a selection in PROMELA. Unions are represented in PROMELA as a series of "`#if ... #endif`" blocks that implement a logical case statement. The blocks use integer identifiers assigned to each module in a file named `Mods.env`. This file is used for the specification of selections in PROMELA.

example, consider the selection

$$(RT\_System\backslash t)[(HLRTP\_LSS\_Pair\backslash x)[LP\_Subservice]].$$

The PROMELA specification of this selection is given in Fig. 10.

The implementation of the function β in SPINe is straightforward. If M is an ordinary module (not a selection), then the file containing the PROMELA description of M is used. If M is a selection, a file specifying the selection is created, for example, as in Fig. 10. Then to discharge $\beta(M) \equiv \beta(N)$, SPINe is invoked with the associated files for M and N. $\beta(M)$ and $\beta(N)$ are computed by the validation engine using a specific state space exploration algorithm [24]. This computation is performed *on the fly* during the equivalence check from the PROMELA descriptions of the two modules.

7. Discussion

This paper presented an approach based on architectural specifications for the expression, derivation, and decomposition of proof obligations of concurrent systems. Here, architecture refers to the structure of a concurrent system and is specified in terms of a simple formalism with a box-and-line type graphical notation. The central feature of the formalism is its ability to express variable systems as groups of modules with identical interfaces.

Proof obligations generated from such specifications are discharged as model checking tasks in a given behavioral model. The approach proposed is independent both of the particular correctness criterion adopted and of the behavioral model in which the proof obligations are discharged. Lowest level architectural components are assigned their respective functionalities in the behavioral model. Since the correctness criterion is based on behavioral equivalence, proof obligations represent equivalence checking tasks. In the examples provided, PROMELA was used as the behavioral model, trace equivalence as the correctness criterion, and an extension to the SPIN tool as the model checker to discharge the proof obligations. In principle, other specification languages and formalisms such as LOTOS [58], CCS [46], Estelle [58], and SDL [58,51] can be used to assign functionality to architectural components. In addition, the block diagram notations of such languages as

SDL and ROOM [52] can be adapted to support system variability. For LOTOS, a box-and-line notation has been proposed in [31] to express process interconnection structures graphically.

An assume-guarantee style reasoning was used in the decomposition of proof obligations. Decomposition required hypothesizing that such assume-guarantee conditions hold for the obligations of the decomposed system. These conditions were captured by approximation relationships among the enclosing contexts (environments) of the modules involved. The problem of checking the validity of approximations was not addressed here. A constructive approach is recommended, whereby approximations (and also abstractions) can be obtained in the behavioral model compositionally from the specifications of the components involved. Such approaches have been proposed in the model checking literature [38,15,12]. These works were discussed briefly in Section 1.1.

The correctness criterion can be changed depending on which properties are of interest and the capabilities of the model checker used. For example, if liveness properties are of interest, failures equivalence [10] or testing equivalence [9] can be adopted as the correctness criterion; if fairness properties are important, ω-language equivalence [38] can be used.

It is also possible to define the correctness criterion in terms of a behavioral preorder rather than an equivalence. A behavioral preorder captures the notion of a concrete system implementing, refining, or simulating an abstract system. Examples can be found in [19,11,9]. In this case, proof obligations would be derived from complete partial orders of modules rather than equivalence classes. The techniques presented for obligation decomposition and reduction can easily be adapted to a preorder-based correctness criterion.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, January 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–534, May 1995.

[3] G. Abowd, , R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. Technical Report CMU-CS-95-111, Carnegie Melon University, School of Computer Science, Pittsburgh, PA, March 1995.

[4] R. Allen and D. Garlan. Beyond definition/use: Architectural interconnection. In *Proc. of Workshop on Interface Definition Languages*, January 1994.

[5] R. Allen and D. Garlan. Formalizing architectural connection. In *Proc. 16th Internat. Conf. on Software Engineering*, pages 71–80, May 1994.

[6] R. Allen and D. Garlan. The Wright architectural specification language. Draft Report CMU-CS-96-xx, Carnegie Melon University, School of Computer Science, Pittsburgh, PA, September 1996.

[7] D. Battory et al. Creating reference architectures: An example in avionics. In *Proc. Symp. on Software Reusability*, 1995.

[8] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced: preliminary report. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, 1988.

[9] E. Brinksma. On the existence of canonical testers. Memorandum INF-87-5, Depart-

ment of Informatics, University of Twente, Netherlands, 1987.

[10] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, number 197 in Lect. Notes Comput. Sci. Springer-Verlag, 1984.

[11] R. Civalero, B. Jonsson, and J. Nilsson. Validating simulations between large nondeterministic specifications. In *Proc. 6th Internat. Conf. on Formal Description Techniques*, pages 3–17, 1993.

[12] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, September 1994.

[13] P. C. Clements. Formal methods in describing architectures. In *Proc. Workshop on Formal Methods in Software Architecture*, 1995.

[14] P. C. Clements. A survey of architecture description languages. In *Proc. 8th Internat. Workshop on Software Specification and Design*, March 1996.

[15] H. De-Leon and E. Grumberg. Modular abstractions for verifying real-time distributed systems. *Formal Methods in System Design*, 2(1):7–43, 1993.

[16] R. De Nicola. Extensional equivalences for transition systems. *Acta Inform.*, 24:211–237, 1987.

[17] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoret. Comput. Sci.*, 34:83–133, 1984.

[18] T.R. Dean and J.R. Cordy. A syntactic theory of software architecture. *IEEE Trans. Softw. Eng.*, 21(4):302–313, April 1995.

[19] D.L. Dill, A.J. Hu, and H. Wong-Toi. Checking for language inclusion using simulation preorders. In *Proc. 3rd Workshop on Computer-Aided Verification*, 1991.

[20] H. Erdogmus. A formal framework for software architectures. Technical Report ERB-1047, National Research Council of Canada, Institute for Information Technology, Ottawa, Ontario, December 1995.

[21] H. Erdogmus. Verifying semantic relations in SPIN. In *Proc. 1st SPIN Workshop*, Verdun, Québec, Canada, October 1995. INRS-Télécommunications.

[22] H. Erdogmus. Architectural specifications, proof obligations, and decomposition. Technical Report, National Research Council of Canada, Institute for Information Technology, Ottawa, Canada, March 1997.

[23] H. Erdogmus. Verification of concurrent systems based on equivalence checking in SPIN. Technical Report ERB-1050, National Research Council of Canada, Institute for Information Technology, Ottawa, Ontario, February 1997.

[24] H. Erdogmus, R. Johnston, and C Cleary. Formal verification based on relation checking in SPIN. In *Proc. 1st Workshop on Formal Methods in Software Practice*, San Diego, CA, January 10–11 1996.

[25] J. Fernandez and L. Mounier. Verifying bisimulations on the fly. In *Proc. 3rd Internat. Conf. on Formal Description Techniques*, 1990.

[26] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *SIGSOFT'94, Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 175–188, December 1994.

[27] D. Garlan, R. Monroe, and D. Wile. ACME: An architectural interchange language. In *Proc. ICSE'97, 19th Internat. Conf. on Software Engineering*, 1997.

[28] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing, 1993.

[29] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994.

[30] M. Hennessy and R. Cleaveland. Testing equivalence as a bisimulation equivalence. *Form. Asp. Comput.*, 5:1–20, 1993.

[31] J. Hinterplattner, H. Nirschl, and H. Saria. Process topology diagrams. In *Proc. 3rd Internat. Conf. on Formal Description Techniques*, pages 535–550, 1990.

[32] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, N.J., 1991.

[33] G. J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks*

and ISDN Systems, 25(9):981–1017, 1993.

[34] P. Inverardi and A.L. Wolf. Formal specification and analysis of software architectures using the Chemical Abstract Machine Model. *IEEE Trans. Softw. Eng.*, 21(4):373–386, April 1995.

[35] ITU. *CCITT Specification and Description Language, Recommendation Z.100.* International Telecommunications Union, 1993.

[36] F. Jananian and A. Mok. Modechart: A specification language for real-time systems. *IEEE Trans. Softw. Eng.*, 21(12), December 1994.

[37] B. Jonsson. Modular verification of asynchronous networks. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 152–166, August 1987.

[38] R. P. Kurshan. Analysis of discrete event coordination. In *Proc. REX Workshop*, number 430 in Lect. Notes Comput. Sci., pages 414–453. Springer-Verlag, 1989.

[39] K. G. Larsen. A context dependent bisimulation between processes. *Theoret. Comput. Sci.*, 49:185–215, 1987.

[40] K. G. Larsen. Efficient local correctness checking. In *Proc. 4th Workshop on Computer-Aided Verification*, pages 35–47, 1992.

[41] G. Leduc. *On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS.* Thèse d'agréation de l'enseignement supérieur, Faculté des sciences appliquées, Université de Liège, Belgium, June 1991.

[42] D. C. Luckham et al. Specification and analysis of system architectures using Rapide. *IEEE Trans. Softw. Eng.*, 21(6), April 1995.

[43] D. C. Luckham and J. Vera. An event based architecture definition language. *IEEE Trans. Softw. Eng.*, 21(9), September 1995.

[44] D. C. Luckham, J. Vera, and S. Meldal. Three concepts of architecture. Technical report, The Program Analysis and Verification Group, Computer Science Department, Stanford University, Stanford, CA, July 1995.

[45] A.N. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th Annual Symp. on Principles of Distributed Computing*, August 1987.

[46] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[47] R. T. Monroe and D. Garlan. Style-based reuse for software architectures. In *Proc. 1996 Internat. Conf. on Software Reuse*, 1996.

[48] M. Moriconi and X. Qian. Correctness and composition of software architecture. In *SIGSOFT'94, Proc. of 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 164–174, December 1994.

[49] M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct architecture refinement. *IEEE Trans. Softw. Eng.*, 21(4):356–372, April 1995.

[50] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[51] A. Sarma. Introduction to SDL-92. *Comput. Netw. ISDN Syst.*, 28(12):1603–1615, June 1996.

[52] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling.* Wiley, 1994.

[53] M. Shaw. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(6), April 1995.

[54] M. Shaw and D. Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, Carnegie Melon University, School of Computer Science, Pittsburgh, PA, December 1994.

[55] M. Shaw and D. Garlan. Formulations and formalisms in software architecture. In *Lect. Notes Comput. Sci.*, volume 1000. Springer-Verlag, 1995.

[56] S.K. Shukla et al. On the complexity of relational problems for finite state processes. In *Proc. Internat. Colloq. on Automata, Languages and Programming*, Paderborn, Germany, July 1996.

[57] I. Sommerville and G. Dean. PCL: A language for modeling evolving software architectures. *Software Eng. J.*, 11(2):111–121, 1996.

[58] K. Turner, editor. *Using Formal Description Techniques: An Introduction to ES-*

TELLE, LOTOS, and SDL. Wiley, 1993.