Modular Completeness: Integrating the Reuse of Specified Software in Top-down Program Development*

Job Zwiers[1], Ulrich Hannemann[2], Yassine Lakhneche[2], Willem-Paul de Roever[2], Frank Stomp[3]

[1] Twente University, P. O. Box 217, 7500 AE Enschede, The Netherlands. Email: zwiers@cs.utwente.nl.
[2] Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Preußerstraße 1-9, 24105 Kiel, Germany. Email: {uha, yl, wpr}@informatik.uni-kiel.d400.de.
[3] AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA. Email: frank@research.att.com.

Abstract. *Reuse* of correctly specified software is crucial in bottom-up program development. Compositional specification formalisms have been designed to reduce the specification of a syntactically composed construct to specifications of its components, and therefore support top-down development methodology. Thus, the integration of reuse of correctly specified software components in a compositional setting calls for adaptation of a given specification to specifications needed in particular circumstances (depending on their application). Proof systems in which such adaptation steps can be performed whenever they are valid are called *modular complete* [Z89]. We present a generic way of constructing such systems for sequential and concurrent Hoare logics.

1 Introduction

Within software development, *reuse* of correctly specified software modules contributes to the efficiency of the programming process while at the same time improving its reliability. Obviously, such reuse belongs to bottom-up program development. Equally obvious, top-down development is supported best by compositional formalisms which have been designed to reduce the specification of a syntactically composed construct to specifications of its components. Thus, their combination with reuse of correctly specified software components calls for an *integrated* specification methodology in which top-down decomposition is combined with bottom-up composition of given specifications for already developed systems, "residing on the shelf of one's software library". The additional property which such an integrated methodology should satisfy is called *the adaptation*

* The collaboration of the authors has been partially supported by the European Community ESPRIT Basic Research Action Project 6021 (REACT).

property [H71]: whenever satisfaction of a fixed, given, specification implies satisfaction of an alternative specification, *this can be deduced within that methodology.* Such formalisms embody the realization of Lamport's concept of a specification *as a contract between a client and the implementation* [L83], because that specification can now be adapted to any context in which such implied specifications are needed, and hence programming with such specifications becomes possible. Compositional formalisms which additionally satisfy this property are called *modular complete* in [Z89] and represent a formal counterpart of such integrated top-down/bottom-up development methodologies. We present a generic way of constructing modularly complete proof systems for Hoare logics, which especially applies to capturing parallelism, and illustrate this technique for Hoare logics in which the interaction between a process and its parallel environment is characterized by a pair of assumption/commitment in [MC81, ZBR83]. The same technique applies to the rely-guarantee formalism of [J81, Q92] and the presupposition/affirmation style of [P88, PJ91].

Recently the interest in modularly complete formal methods has been revived. In [deB94] a semantic analysis of such logics in compositional settings for distributed communication and shared variable parallelism is presented, [M94] identifies a subset of his temporal interval logic formalism as modular complete, and in [AL94] such a formalism is developed for Lamport's TLA [L91]. Ramesh [R90] also studies the notion of modular *completeness.*

Technically the problem of adaptation is a fundamental one, and emerges already in the context of Hoare style pre/post specifications, where transitions are characterized by pairs of predicates. Where top-down development meets bottom-up, there is in general a gap between a required specification $(\tilde{pre}, \tilde{post})$ and a provided specification $(pre, post)$ in such formalisms. For instance, the bottom up specification might be a translation of a VDM or Z style specification. In that case, the $(pre, post)$ specification is of the form $(\bar{x} = \bar{v}, post(\bar{v}, \bar{x}))$, thus relating the program variables $\bar{x}$ explicitly to so-called *freeze* variables $\bar{v}$. Most likely this does not directly match the required $(\tilde{pre}, \tilde{post})$ specification, and one needs proof rules like the rule of consequence [H69] as well as various substitution rules [G75] in order to derive $(\tilde{pre}, \tilde{post})$ from $(pre, post)$. The adaptation rule from [H71], and also the stronger rules proposed in [O83, D92], look rather complicated, certainly when compared to other rules of Hoare's logic (These adaptation rules apply to *arbitrary* preconditions, whereas [G75] is limited to preconditions of the form $\bar{x} = \bar{v}$.) The same remark applies to the adaptation rules put forward in this paper. Now the way we arrive at these adaptation rules indicates an alternative way to achieve adaptation. The idea is to switch back and forth between a given specification formalism and an alternative formalism, with adaptation carried out in the latter one. In order to do so, we propose here rather simple rules for translating back and forth between Hoare's logic and single predicate specifications in the style of VDM or Z. Adaptation of single predicates boils down to showing logical implication between two predicates, which is easier than adaptation of Hoare style formulae.

We present similar translation rules for the assumption/commitment framework. Here we translate specifications that consist of a pre/post condition together with an assumption/commitment pair to single predicates on traces. Again this reduces the adaptation problem to showing logical implication between predicate logic formulae.

Given the complexity of these adaptation rules, one might wonder whether it wouldn't make sense to stick to single predicate specification formalisms such as VDM. The problem is that such formalisms don't cope conveniently with sequential constructs, and that application of their parallel composition rules requires complicated induction arguments which are rendered superfluous in case of assumption/commitment and rely/guarantee based formalisms.

Section 2 describes a single, unified formalism which is essentially a variation of second order predicate logic. In Sect. 3 a generic solution of the adaptation problem is presented in a sequential setting, which is extended in Sect. 4 to parallelism. Section 5 draws some conclusions.

2 A Uniform Framework

We discuss specification adaptation and modularity for a variety of specification formalisms, both for sequential and concurrent systems. Think of specification of states, specification of state transitions, specification of communication histories etc. Despite this variety of applications we have a single, unified, logic formalism which is essentially a variation of second order predicate logic. Languages for programs, mixed terms, pre/postconditions and correctness formulae are defined as sub-languages of the unified language. One of the advantages of using a unified formalism is that it becomes possible to translate back and forth between sub-languages, where the translation can be carried out fully within the formalism. This is used to ease the adaptation of specifications. A second advantage is that it becomes easier to relate our results to other formalisms based on logic, such as TLA [L91], VDM [J86], Z [S92]. After defining the predicate logic we show how the embedding works out for VDM, Z and Hoare style pre-postcondition specifications, and for trace based CSP specifications.

2.1 Predicate Formulae

We assume given sets of typed first order variables $x \in Var$, and of (second order) relation variables $X \in VAR$. The exact typing scheme is not important for our goal; we only assume that each variable x has an associated type τ and that the type of a relation variable is a pair of lists of first order variables. Variables x can be *decorated*, for instance by means of primes like x', x'', $\overleftarrow{x}$. Such decorated versions are considered to be different variables, of the same type as the undecorated version. Expressions e are built from first order variables x and constants c by means of operations f, where appropriate typing constraints are to be observed, as usual. Relation constants and relation variables are interpreted as $n - ary$ typed relations, where both an arity and a typing are assumed to be associated with each constant R and variable X. We assume that among these relational constants we have at least the equality relation "=", for each first order type τ. Within the syntax of formulae below, it is tacitly assumed that typing constraints are satisfied.

The class of predicate formulae $\phi \in Pred$ is defined by:

$$\phi ::= e_0 = e_1 \mid R(e_0, \ldots, e_{n-1}) \mid X(e_0, \ldots, e_{n-1}) \mid \phi_0 \wedge \phi_1 \mid \neg\phi \mid \forall x.(\phi) \mid \forall X.(\phi)$$

Formulae like $\phi_0 \rightarrow \phi_1$, where $\rightarrow$ denotes implication, are seen as standard abbreviations of *Pred* formulae, and will be used freely. $FV(\phi)$ denotes the free first order variables of ϕ. A more substantial abbreviation that we use is $\mu X.\phi(X)$, denoting the smallest predicate Y such that $\phi(Y)$ is valid.

2.2 The Satisfaction Relation

Below we explain how to embed VDM, Z, Hoare style formulae and trace based CSP specifications in the predicate logic. All these embeddings have in common is that they are based on a general *satisfaction relation* "**sat**". A formula of the form ϕ **sat** ψ is defined here as a predicate formula $\forall \bar{x}.(\phi \rightarrow \psi)$. The variables $\bar{x}$ are the so-called *base variables*, denoted by $base(\phi, \psi)$, of the specifications ϕ and ψ; they are the subset of $FV(\phi, \psi)$ consisting of those variables that refer to the *observable behavior*. For typical sequential systems, the base of a system or specification is a set of state variables, possibly decorated to distinguish between initial and final state values. On the other hand, trace based specifications for communicating processes are predicates with a single base variable "h" that denotes the communication history of a process. Ready-trace specifications for CSP have a base consisting of a history "h" and a ready set "R". In [Z89] the base of process specifications includes a history h as well as (decorated) local state variables.

The remaining first order variables, i.e. the non-base ones, are denoted by $lvar(\phi)$, and are called the *logical variables* of ϕ; in the literature one also refers to these as "freeze variables", or "rigid variables". We assume that syntactic conventions are used to distinguish base variables from logical variables. When relation variable Y occurs within a system S with base $\{\bar{x}, \bar{x}'\}$, then Y without explicit arguments abbreviates $Y(\bar{x}, \bar{x}')$.

2.3 Decorated Identifiers

Sequential programs and *state transitions* are relations on states; relations in turn are regarded as predicate formulae where certain variables relate to inputs and other variables relate to outputs. Together these variables constitute the *base* of a specification. In concrete specification languages one employs declarations and suitable syntactic conventions for indicating which variables x are inputs and which are outputs, usually by means of *decorated identifiers* like $\overleftarrow{id}$, id', or id°. Here we only assume that $base(\phi)$ is the disjoint union of *input variables* $in(\phi)$, *output variables* $out(\phi)$. If x is some state variable then we use $\underline{x}$ to denote the undecorated version of x, x' to denote the corresponding output version, and $`x$

to denote the corresponding input version. Note that we employ $\underline{x}$, ${}^{\backprime}x$ and x' not as decorations as such but rather as meta operations on decorated identifiers, that will work out differently for particular formalisms. A few concrete examples:

- For VDM specifications, undecorated identifiers id denote outputs, whereas "hooked" identifiers like $\overleftarrow{id}$ denote inputs. So for this case, $\underline{(\overleftarrow{id})}$ as well as $(\overleftarrow{id})'$ equal id, and ${}^{\backprime}(id)$ is $\overleftarrow{id}$.
- For Z schemes and TLA formulae, undecorated identifiers id denote inputs, whereas "primed" identifiers like id' denote outputs. For this case $(id)'$ is id', and $\underline{(id')}$ as well as ${}^{\backprime}(id')$ equal id.
- In [He84], our notation ${}^{\backprime}x$ and x' is employed, not on the meta level but rather as actual decoration for input and output variables. So, with slight abuse of notation, $({}^{\backprime}(id))'$ is id' and ${}^{\backprime}(id')$ is ${}^{\backprime}id$.

Decoration applied to predicates ϕ denotes that all relevant variables occurring free in ϕ are decorated.

2.4 Sequential Programs

Sequential *programs* can be translated into predicates, too. That is, assume that we have a program S that reads and writes a set of state variables $\bar{x}$. (One says that S is based on $\bar{x}$.) Then one can construct a predicate formulae ϕ_S with base ${}^{\backprime}\bar{x}, \bar{x}'$ that captures the input-output behavior of S. The translation can be given in a compositional style. As an example, if programs S and T can be translated into ϕ_S and ϕ_T, then $S\,;T$ can be translated into: $\phi_S\,;\,\phi_T \stackrel{\text{def}}{=} \exists\bar{z}.(\phi_S[\bar{z}/\bar{x}'] \wedge \phi_T[\bar{z}/{}^{\backprime}\bar{z}])$, where $\bar{z}$ is a list of undecorated identifiers such that $out(\phi_S) \subseteq \{\bar{z}'\}$, and $in(\phi_T) \subseteq \{{}^{\backprime}\bar{z}\}$. In the sequel, we treat programs as (abbreviations of) predicate formulae.

2.5 Predicate Transformers and Hoare's Logic

Let p, q be predicates with $base(p, q) \subseteq \{\bar{x}\}$, and let S be a predicate with $base(S) \subseteq \{{}^{\backprime}\bar{x}, \bar{x}'\}$. Let ${}^{\backprime}p$ denote the predicate $p[{}^{\backprime}\bar{x}/\bar{x}]$ and let q' denote the predicate $p[\bar{x}'/\bar{x}]$. We introduce Hoare formulae, weakest preconditions and strongest postconditions as abbreviations:

$$\{p\}\ S\ \{q\} \text{ abbreviates: } S \textbf{ sat } ({}^{\backprime}p \rightarrow q').$$

$$wp(S, q) \text{ abbreviates: } \underline{\forall \bar{x}'.(S \rightarrow q')}$$

$$sp(p, S) \text{ abbreviates: } \underline{p'\,;\,S} \text{ or, equivalently: } \underline{\exists {}^{\backprime}\bar{x}.({}^{\backprime}p \wedge S)}$$

Note that the definitions are not limited to predicates S that result from translating a program text. So one can, for instance, calculate $wp({}^{\backprime}p \rightarrow q', r)$. This aspect is of vital importance for our results on modular completeness for Hoare's logic.

2.6 Trace Logic and CSP

Trace based specifications are assertions of the form $S(h)$ with free occurrences of a trace typed variable h. A *trace* is a finite sequence of communications, as usual [CH81]. We employ standard notations and operations for traces, such as *concatenation* $t_0 \,\hat{}\, t_1$ of traces t_0 and t_1, and *projection* $t \,|\, \alpha$ of trace t onto alphabet α. For a non-empty trace t, i.e. of the form $t_f \,\hat{}\, \langle a \rangle$, we denote communication a by $last(t)$ and the remaining sequence t_f by $rest(t)$. The special communication symbol "$\surd$", is used to signal that a process has terminated. We define *sequential composition* $t_0 \,;\, t_1 = t_0$ if $last(t_0) \neq \surd$, and $t_0 \,;\, t_1 = rest(t_0) \,\hat{}\, t_1$ if $last(t_0) = \surd$.

It is known from the literature [CH81, Z89] how to assign to a CSP process a trace specification $S(h)$. Within our uniform framework we can thus regard CSP processes as abbreviations of certain trace specifications. The special variable h is the only base variable of such process specification, i.e. all other free variables are logical variables. This implies that a refinement relation of the form "P **sat** Q" must be read as "$\forall h.(P(h) \rightarrow Q(h))$". Depending on whether P and Q are structured as CSP processes or as logical formulae, we call "**sat**" a *refinement relation*, an *implementation relation*, or a *specification adaptation.*

Composition operations for processes fit easily in this approach. As an example, sequential composition $P \,;\, Q$ is defined by: $(P \,;\, Q)(h) = \exists t_0 \exists t_1.(P(t_0) \wedge Q(t_1) \wedge h = t_0 \,;\, t_1)$.

3 Modularity and Specification Adaptation

We now present a generic solution of the adaptation problem in a sequential setting.

3.1 Specification Adaptation: The Problem

Modular design of systems is often a combination of top-down global design and bottom-up reuse of existing modules. A pure top-down approach starts with a first specification S_0, which is then transformed gradually, via a series of intermediate designs $S_1, \ldots S_{n-1}$, into a final program text S_n. An intermediate design, say S_i, is built up from a number of (logical) specifications $\phi_0, \ldots, \phi_m$ by means of programming language constructs, such as sequential composition and iteration. Since they combine programming constructs with logical specifications, the S_i are called *mixed terms*. Mixed term S_i is obtained from S_{i-1} by replacing some sub-term T in S_{i-1} by an implementing mixed term R, i.e. R must be such that R **sat** T is the case. When T actually has the form of a logical specification and R the form of a program, then this describes a classical, top-down development step. When both T and R are programs, then we have a classical program transformation step.

Finally, when both T and R are logical specifications, we are dealing with specification adaptation. Such an adaptation step in the development can become necessary when one would like to implement by some already available module M, where M is known to satisfy specification R. We do not want to verify *directly* that M **sat** T, as this would force us to inspect the internal structure of M. Our programming language might even include encapsulation constructs that does not allow us to inspect this internal structure. Therefore, we have to check indirectly that M **sat** T, by showing that R **sat** T. Because T and R are specifications created by different designers, there might be a substantial "gap" between the two, and consequently verifying that R **sat** T then becomes a non-trivial design step. This is especially the case when at least one of T and R is a pre/post specification.

3.2 Specification Adaptation for Pre/post Specifications

We consider the case where both are pre/post specifications: Assume that T is determined by a pre/post condition pair $pre_T, post_T$, that is, we seek a module "X" such that $\forall\bar{v}.(\{pre_T\}\ X\ \{post_T\})$, where $\bar{v}$ are the logical variables of $pre_T, post_T$. Assume that R is a similar pair of assertions $pre_R, post_R$, with logical variables $lvar(pre_R, post_R) = \bar{u}$. The check "$R$ **sat** T" now boils down to checking that $\forall\bar{u}.(\{pre_R\}\ X\ \{post_R\})$ implies $\forall\bar{v}.(\{pre_T\}\ X\ \{post_T\})$.

This is the problem of specification adaptation, for the case of pre/post specifications. Already in [G75] this problem was considered for proof systems for (parameterless) recursive procedures. In this case pre_R is restricted to be of the form $\bar{x} = \bar{u}$. (A so-called "freeze" predicate.) It was shown that the following set of rules suffices, where we have reformulated the rules in [G75] to fit in our framework:

$$\frac{\tilde{p}\rightarrow p,\ \{p\}\ S\ \{q\},\ q\rightarrow\tilde{q}}{\{\tilde{p}\}\ S\ \{\tilde{q}\}} \qquad \text{(Rule of consequence)}$$

$$\frac{\forall v.(\{p\}\ S\ \{q\})}{\forall u.((\{p[u/v]\}\ S\ \{q[u/v]\})},\ \text{provided } v \text{ not free in } S \qquad \text{(Substitution)}$$

$$\frac{\{p_i\}\ S\ \{q_i\},\ \text{for } i = 0, 1}{\{p_0 \wedge p_1\}\ S\ \{q_0 \wedge q_1\}} \qquad \text{(Conjunction)}$$

$$\frac{\{p(v)\}\ S\ \{q\}}{\{\exists v.(p(v))\}\ S\ \{q\}},\ \text{provided } v \text{ not free in } q, S \qquad \text{(Elimination)}$$

$$\{p\}\ S\ \{p\} \qquad \text{(Invariance)},$$

provided that p contains no free occurrences of state variables. (Only logical variables are allowed.)

An interesting alternative to this set of rules are the so-called *rules of adaptation*, proposed by Hoare [H71], Dahl [D92], and Olderog [O83]. Each of these three rules can, together with the rule of consequence, replace the set of rules proposed by [G75]. We have listed the rules below, where we have used explicit quantifiers for logical variables around Hoare formulae. (Within pure Hoare logics such quantifiers are left implicit). Let $\{\bar{x}\} = base(p, q, r)$, $\{\bar{u}\} = lvar(p, q)$,

$\{\bar{v}\} = \mathit{lvar}(r)$, $\{\bar{z}\} = \{\bar{u}\} - \{\bar{v}\}$, $\{\bar{w}\} = \{\bar{v}\} - \{\bar{u}\}$, and assume that $\mathit{lvar}(S) \cap \mathit{lvar}(p,q,r) = \emptyset$, and let $\tilde{u}$ be a fresh list of logical variables

$$\frac{\forall\bar{u}.(\{p\}\ S\ \{q\})}{\forall\bar{w}.(\{\exists\bar{z}.(p \wedge \forall\bar{x}.(q \rightarrow r))\}\ S\ \{r\})} \qquad \text{(Hoare's rule of adaptation)}$$

$$\frac{\forall\bar{u}.(\{p\}\ S\ \{q\})}{\forall\bar{v}.(\{\exists\tilde{u}.(p[\tilde{u}/\bar{u}] \wedge \forall\bar{x}.(q[\tilde{u}/\bar{u}] \rightarrow r))\}\ S\ \{r\})} \qquad \text{(Dahl's rule of adaptation)}$$

$$\frac{\forall\bar{u}.(\{p\}\ S\ \{q\})}{\forall\bar{v}.(\{\forall\bar{x}'.(\forall\bar{u}.({}^{\backprime}p \rightarrow q') \rightarrow r')\}\ S\ \{r\})} \qquad \text{(Olderog's rule of adaptation)}$$

As analyzed in [O83], the precondition in Olderog's rule is actually the weakest precondition $wp(\forall\bar{u}.({}^{\backprime}p \rightarrow q'), r)$. As pointed out in [O83] the precondition in Hoare's rule is not the weakest precondition. As pointed out by [D92], the precondition in Dahl's rule, although weaker than the Hoare's, appears to be the weakest precondition only when a total correctness interpretation for Hoare formulae is assumed.

We present a fourth rule of adaptation, based on the observation that the strongest postcondition $sp(r, \forall\bar{u}.({}^{\backprime}p \rightarrow q'))$ is equivalent to the predicate formula $\exists{}^{\backprime}\bar{x}({}^{\backprime}r \wedge \forall\bar{u}({}^{\backprime}p \rightarrow q))$.

$$\frac{\forall\bar{u}.(\{p\}S\{q\})}{\forall\bar{v}.(\{r\}S\{\exists{}^{\backprime}\bar{x}({}^{\backprime}r \wedge \forall\bar{u}({}^{\backprime}p \rightarrow q))\})} \qquad \text{(SP version of rule of adaptation)}$$

The rules of adaptation are somewhat complicated to work with. We therefore propose an alternative set of adaptation rules that cannot be formulated within Hoare logics. Rather they allow one to switch between Hoare style pre/post specifications and VDM or Z style specifications. By switching back and forth, one obtains Olderog's rule of adaptation or our SP version as derived rules. Again we have formulated the rules with explicit quantifiers for logical variables $\bar{u}, \bar{v}$ around Hoare formulae.

$$\frac{\forall\bar{u}.(\{p\}\ S\ \{q\})}{S\ \mathbf{sat}\ \forall\bar{u}.({}^{\backprime}p \rightarrow q')} \qquad \text{(Hoare-SAT)}$$

$$\frac{S\ \mathbf{sat}\ \psi}{\forall\bar{v}.(\{wp(\psi, r)\}\ S\ \{r\})} \quad \text{provided } \bar{v} \text{ not free in } S \qquad \text{(SAT-WP)}$$

$$\frac{S\ \mathbf{sat}\ \psi}{\forall\bar{v}.(\{r\}\ S\ \{sp(r, \psi)\})} \quad \text{provided } \bar{v} \text{ not free in } S \qquad \text{(SAT-SP)}$$

Actually, the Hoare-SAT rule is an equivalence, that is, $\forall\bar{u}.(\{p\}\ S\ \{q\})$ iff S **sat** $\forall\bar{u}.({}^{\backprime}p \rightarrow q')$. This suggests (yet) another way for adaptation of Hoare formulae: If $\forall\bar{u}.({}^{\backprime}p \rightarrow q')$ implies $\forall\bar{v}.({}^{\backprime}r \rightarrow t')$, then from $\forall\bar{u}.(\{p\}\ S\ \{q\})$ it follows that $\forall\bar{v}.(\{r\}\ S\ \{t\})$. A Hoare style rule along these lines was formulated by Cartwright and Oppen [CO81, O83].

3.3 Modular Completeness

The classical notion of (relative) completeness does not suffice for proof systems aiming at modular verification. This sort of completeness, that we will call *compositional completeness* of a given proof system asserts the following: Assume we have a structured system S, of the form $C(S_1, \ldots, S_n)$ where $n \geq 0$. If $C(S_1, \ldots, S_n)$ **sat** *spec* is a *valid* formula, then *there exist* specifications $spec_i$, such that S_i **sat** $spec_i$ is valid for $1 \leq i \leq n$, and moreover, the proof system allows a *derivation* of $C(S_1, \ldots, S_n)$ **sat** *spec* from the premisses S_i **sat** $spec_i, 1 \leq i \leq n$. Note that the specifications of the S_i can be chosen such as to suit the derivation. For this reason, compositional completeness is the appropriate completeness notion for *top-down* development.

A stronger notion is *modular completeness*. Here one considers systems of the form $S(X_1, \ldots, X_n)$, where the X_i represent modules with a priori given specifications. The idea is that the X_i will be replaced by implementations S_i that satisfy these specifications, but we may neither inspect the internals of the S_i, nor can we design or otherwise influence the given specifications. Modular completeness asserts the following: If $(\bigwedge_{i=1}^{n} X_i \textbf{ sat } spec_i) \rightarrow S(X_1, \ldots, X_n)$ **sat** *spec* is a *valid* formula, the proof system allows a *derivation* of $S(X_1, \ldots, X_n)$ **sat** *spec* from the premisses X_i **sat** $spec_i$. In full generality, we also allow for the case where one module X_i has several (complementary) specifications. We call a proof system *strong adaptation complete* if whenever a formula of the form

$$\Big(\bigwedge_{1 \leq j \leq n} X_{i_j} \textbf{ sat } spec_j\Big) \rightarrow X \textbf{ sat } spec$$

is valid, then it is *derivable* within the proof system. In this definition, X_{i_j} is the variable Y such that X_j and Y are syntactically the same, $1 \leq i_j \leq n$, and X is any of the variables $X_1, \ldots, X_n$. *Adaptation completeness* is defined as strong adaptation completeness but with $n = 1$.

Lemma 1. *A proof system that is both compositionally complete and strong adaptation complete is modularly complete.*

The adaptation rules for Hoare's logic presented in previous sections aim at adaptation completeness. It is easy to change them into rules that achieve *strong* adaptation completeness. For instance, our SP-adaptation rule becomes:

$$\frac{\bigwedge_{i=1}^{n} \forall \bar{u}.(\{p_i\}S\{q_i\})}{\forall \bar{v}.(\{r\}S\{\exists \grave{\bar{x}}(\grave{r} \wedge \forall \bar{u}(\bigwedge_{i=1}^{n}(\grave{p}_i \rightarrow q_i))))\}} \qquad \text{(Strong SP adaptation)}$$

We end this section with two lemmata which formalize the relationship between modular completeness and top-down program development.

Lemma 2. *Let $S(X_1, \cdots, X_n)$ be a system. Consider pairwise distinct variables $Y_1, \cdots, Y_m$ all in $\{X_1, \cdots, X_n\}$, and sets of indices $I_j = \{i \mid 1 \leq i \leq n$, and Y_j and X_i denote the same variable$\}$, $j = 1, \cdots, m$. If the formula $\bigwedge_{1 \leq i \leq n} X_i$ **sat** $spec_i$ is valid, then (a) and (b) are equivalent:*

(a) *$S(X_1, \cdots, X_n)$ **sat** spec is valid.*

(b) *For all formulae $S_1, \cdots, S_m$ whose relation variables are all among $X_1, \cdots, X_n$, the formula $((\bigwedge_{1 \leq j \leq m} \bigwedge_{k \in I_j} S_k$ **sat** $spec_j) \rightarrow S'$ **sat** spec) is valid. Here, S' is the formula obtained from $S(X_1, \cdots, X_n)$ by substituting S_j for every variable Y_j, $1 \leq j \leq m$.*

As an immediate consequence we of this lemma we obtain its proof theoretic counterpart:

Corollary 3. *Let T be a modularly complete proof system. Using the same notation as in the previous lemma, and assuming that $\bigwedge_{1\leq i\leq n} X_i$* **sat** *$spec_i$ is derivable in T, (a) and (b) are equivalent:*

(a) S **sat** *$spec$ is derivable in T.*
(b) For all formulae $S_1, \cdots, S_m$ whose relation variables are all among $X_1, \cdots, X_n$, $((\bigwedge_{1\leq j\leq m} \bigwedge_{k\in I_j} S_k$ **sat** *$spec_j) \rightarrow S'$* **sat** *$spec)$ is derivable in T.*

4 Modular Completeness for Concurrent Processes

For CSP processes various specification and verification styles are known. We discuss some of these, in particular the issue of specification adaptation for such formal systems.

4.1 "SAT" Proof Systems and Generalized Hoare Logic

Simple proof systems have been built around the "**sat**" relation. [H71, Z89, O83] Specification adaptation is rather straightforward here, which is certainly one of the advantages of such "SAT" systems: Let $P(h)$ and $Q(h)$ be logical specifications. Then it is clear that $\forall X.((X \textbf{ sat } P)\rightarrow(X \textbf{ sat } Q))$ iff $P \textbf{ sat } Q$. The latter formula abbreviates $\forall h.(P(h)\rightarrow Q(h))$, which is a simple verification condition. Proof rules for "**sat**" formulae are easily formulated, but the rules for sequential constructs are not very helpful. For example, in the conclusion of:

$$\frac{P_0 \textbf{ sat } Q_0\ ,\ P_1 \textbf{ sat } Q_1}{P_0\ ;\ P_1 \textbf{ sat } Q_0\ ;\ Q_1} \qquad \text{(Sequential Composition)}$$

the specification $Q_0\ ;\ Q_1$ still contains a sequential operator, that can be eliminated only at the expense of introducing an existential quantifier. A "nicer" rule can be obtained by mimicking Hoare's logic within the SAT system. For any trace predicate S, define a *relation* on traces " ; S" as follows:

$$(\ ;\ S)(h,h') = \exists h''.(S(h'') \wedge h' = h\ ;\ h'').$$

Note that ; S, as any relation, can be specified by Hoare formulae or predicate transformers. A novel aspect is that within $\{P\}\ ;\ S\ \{Q\}$ the pre and postconditions P and Q are *trace* specifications and so, could be CSP processes themselves. The Hoare formula above is equivalent to the "**sat**" formula $P\,;\,S \textbf{ sat } Q$; a correspondence between a SAT formula and a Hoare formula that resembles the concept of *weakest prespecifications* [HHS87]. Simplicity of this specification style shows up clearly with the rules for sequential composition and iteration; for example:

$$\frac{\{P\}\ ;\ S_0\ \{R\}, \{R\}\ ;\ S_1\ \{Q\}}{\{P\}\ ;\ (S_1\ ;\ S_2)\ \{Q\}} \qquad \text{(Sequential composition)}$$

The adaptation rules for Hoare's logic apply here too. Yet it appears simpler to *specify* reusable modules by means of **sat** specifications. Then, when placed within a sequential context, a **sat** specification S *sat* T can be adapted to a Hoare formulae $\{P\}$; S $\{P\ ;\ T\}$, for any precondition P.

4.2 Extensions of the Hoare Style System for CSP Processes

For practical verification purposes, the Hoare style system for CSP is still not very convenient. Consider a formula $\{P\}\ S\ \{Q\}$. The postcondition Q is required to hold for all traces of the system P ; S, that is, both for traces that end in a "$\surd$" and for those that do not. Intuitively, the traces ending with a "$\surd$" correspond to executions in which control is after process S, whereas traces without such a "tick" correspond to intermediate stages of the execution of process S. In [ZRE84] a class of formulae was introduced where pre/postconditions describe only traces ending with "$\surd$", corresponding to snapshots before and after execution of S, and where the specification of traces without "$\surd$" is delegated to a *trace invariant* I. Let "FIN" be the predicate "$last(h) = \surd$". Then we define $(I\ :\ \{P\}\ S\ \{Q\})$ as an abbreviation for $\{I\wedge(FIN \rightarrow P)\}$; S $\{I\wedge(FIN \rightarrow Q)\}$.

This specification style separates local conditions, such as pre- and postcondition, from the interface towards the entire system without losing the simplicity of the rules of Hoare's logic. Moreover, we can restrict ourselves to assertions for I, P and Q, where these special FIN predicates do not occur anymore in contrast to the Hoare style where they are inevitable [Z89]. Typical proof rules in this style are (details concerning side conditions omitted):

$$\frac{I_0 : \{P_0\}\ S_0\ \{Q_0\},\ I_1 : \{P_1\}\ S_1\ \{Q_1\}}{I_0 \wedge I_1 : \{P_0 \wedge P_1\}\ S_0 \| S_1\ \{Q_0 \wedge Q_1\}} \quad \text{(provided } chanbase(I_i, Q_i) \subseteq \alpha(S_i))$$

$$\frac{I : \{P\} S_0 \{R\}\ ,\ \{R\} S_1 \{Q\}}{I : \{P\} S_0\ ;\ S_1 \{Q\}}$$

An invariant formula $I : \{\ P\ \}\ S\ \{\ Q\ \}$ is equivalent to S **sat** $I : P \leadsto Q$, where $(I : P \leadsto Q)(h)\ \equiv\ \forall t.((FIN(t) \wedge P(t)) \rightarrow (I(t\ ;\ h) \wedge (FIN(t\ ;\ h) \rightarrow Q(t\ ;\ h))))$
This allows for a straightforward adaptation rule:

$$(I : \{P\}\ X\ \{Q\})\ \rightarrow\ (J : \{R\}\ X\ \{T\}) \quad \text{iff} \quad (I : P \leadsto Q)\ \mathbf{sat}\ (J : R \leadsto T).$$

It is possible to present a direct adaptation rule, without switching to "**sat**" formulae, along similar lines as for the case of Hoare formulae:

$$\frac{I : \{\ P\ \}\ S\ \{\ Q\ \}}{sinv(R, (I : P \leadsto Q)) : \{\ R\ \}\ S\ \{\ sp(R, (I : P \leadsto Q))\ \}} \qquad \text{(Adaptation)}$$

where $sinv(R, (I\!: P \leadsto Q)) = R; ((I\!: P \leadsto Q)[\mathbf{true}/FIN(t),\ \mathbf{false}/FIN(t;\ h)])$
$sp(R, (I : P \leadsto Q))\ = R\ ; ((I : P \leadsto Q)[\mathbf{true}/FIN(t),\ \mathbf{true}/FIN(t\ ;\ h)])$.

4.3 Misra/Chandy Specification Style

An *Assumption/Commitment* style was introduced by Misra and Chandy [MC81] to ease inductive forms of reasoning for distributed processes. An *assumption* A refers to the expected communication behavior of the environment and a *commitment* C refers to the communications of the specified module. Proof rules for a state-trace based model were given in [ZBR83] and [ZRE84] as derivations of rules of the invariant system. A compositional complete proof system in this Assumption/Commitment style has been given in [P88], called *P-A Logic*. Here, we introduce assumption/commitment pairs as abbreviations:

Definition 4 A/C Invariants. Let A, C be trace assertions. We abbreviate $(h \neq \langle\rangle \rightarrow A(rest(h)))$ by $\bullet A(h)$; $\forall h'(h' \leq h.(P(h')))$ by $\mathcal{K}ern(P)(h)$; and $\mathcal{K}ern(\mathcal{K}ern(\bullet A(h)) \rightarrow C(h))(h)$ by $(A,\ C)(h)$.

An assumption/commitment specification is a formula of the form: (A, C) : $\{P\}\ S\ \{Q\}$. Since this is just a special case of the formulae we discussed above, rules like the one for parallel composition remain valid. For example, if we have specifications of the form (A_i, C_i) : $\{P_i\}$ S_i $\{Q_i\}$, for $i = 0, 1$, then the rule yields a specification for $S_0 \parallel S_1$ of the form:

$$(A_0, C_0) \wedge (A_1, C_1)\ :\ \{P_0 \wedge P_1\}\ S_0 \parallel S_1\ \{Q_0 \wedge Q_1\}.$$

The invariant of this formula can be rewritten into assumption/commitment style again provided that, for some assertion A, $(A \wedge C_i) \rightarrow A_{i-1}$ for $i = 0, 1$. For, under these assumptions, a proof by induction on the length of traces shows that $((A_0, C_0) \wedge (A_1, C_1))$ **sat** $(A, C_0 \wedge C_1)$.

It will be clear that *adaptation* of assumption/commitment formulae can be achieved by translating such formulae to corresponding "**sat**" formulae [Ha94]. Finally, one can even formulate a, rather complex, adaptation rule within the Assumption/Commitment formalism itself:

Rule 1 (Adaptation Rule)

$$\frac{(A, C) : \{\,P\,\}\, S\, \{\,Q\,\}}{\begin{array}{c}(A',\ \bullet(\mathcal{K}ern(A')) \wedge (sinv(R, ((A, C) : P \rightsquigarrow Q)))) : \\ \{\,R\,\}\ S\ \{\,\mathcal{K}ern(\mathcal{K}ern(A')) \wedge sp(R, ((A, C) : P \rightsquigarrow Q))\,\}\end{array}}$$

The derivation of this rule within the assumption/commitment framework follows the same pattern as before as indicated by the close relationship to the invariant system.

5 Conclusion

We have shown how to approach the problem of specification adaptation for a variety of formal proof systems, both for sequential and for parallel programs. Basically, there are two approaches:

1. Formulate special adaptation rules *within the formal system itself*
2. Formulate transformation rules for *switching back and forth between the given formalism and other, more basic, formalisms.* Adaptation is then carried out in the more basic formalism.

We have illustrated our theory for the sequential case by showing how to translate between Hoare logic, predicate transformers and VDM or Z style specifications. For CSP processes we reduced Misra/Chandy style specifications to invariant formulae, then invariant formulae to (generalized) Hoare formulae, and finally Hoare formulae to SAT formulae. It is interesting to see that on the most basic level, both sequential and concurrent processes can be formalized in essentially the *same* SAT formalism. Moreover, the adaptation problem for SAT systems boils down to showing logical implication between predicates.

To carry out our programme we relied on a *uniform logical framework.* This framework makes it easy to combine, and to translate between, different formalisms.

References

[AL94] Abadi, M., and Lamport L.: *Conjoining Specifications*, DEC Systems Research Center, Research report (1994)

[A81] Apt, K. R.: *Ten Years of Hoare's Logic, A Survey, Part I*, ACM Transactions on Programming Languages and Systems **3**:4 (1981) 431–483

[deB94] de Boer, F.: *Compositionality in the Inductive Assertion Method for Concurrent Systems*, IFIP TC 2 Working Conference on programming concepts, methods and calculi (1994)

[CO81] Cartwright, R., and Oppen, D.: *The logic of aliasing*, Acta Informatica **15** (1981) 365–384

[CH81] Chen, Z. C., and Hoare, C. A. R.: *Partial correctness of CSP*, Conf. on Distr. Comp. Sys. (1981)

[D92] Dahl Ole-Johan: *Verifiable Programming*, Prentice Hall (1992)

[D76] Dijkstra, E. W.: *A discipline of programming*, Prentice-Hall (1976)

[G94] Gibbs, W. W.: *Software's Chronic Crisis* Scientific American **9** (1994)

[GM93] Gordon, M. J. C., and Melham, T. F.: *Introduction to HOL – A theorem proving environment for higher order logic*, Cambridge University Press (1993)

[G75] Gorelick, G. A.: *A complete axiomatic system for proving assertions about recursive programs and non-recursive programs*, TR **75**, University of Toronto (1975)

[H69] Hoare, C. A. R.: *The axiomatic basis of programming*, Communications of the ACM (1969)

[H71] Hoare, C. A. R.: *Procedures and parameters: An axiomatic approach*, Lecture Notes in Mathematics (1971) 102–116

[HHS87] Hoare, C. A. R., He Jifeng, and Sanders, J. W.: *Prespecification in Data Refinement*, Information Processing Letters **25** (1987)

[He84] Hehner, E. C. R.: *Predicative Programming, part I and II*, Communications of the ACM **27** (1984)

[Ha94] Hannemann, U.: *Modular complete proof systems for distributed processes*, M. Sc. thesis, University of Kiel (1994)

[J81] Jones, Cliff B.: *Development methods for computer programs including a notion of interference*, Oxford (1981)

[J86] Jones, Cliff B.: *Systematic software development using VDM*, Prentice-Hall (1986)

[L83] Lamport, L.: *Specifying concurrent program modules*, ACM Transactions on Programming Languages and Systems **6**(2) (1983)

[L91] Lamport, L.: *The Temporal Logic of Actions*, DEC Systems Research Center (1991)

[MP91] Manna, Z., and Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems*, Springer Verlag (1991)

[M88] Meyer, B.: *Object-Oriented Software Construction*, Prentice-Hall (1988)

[MC81] Misra, J., and Chandy, K. M.: *Proofs of networks of processes*, IEEE Transactions on Software Engineering **7**:4 (1981)

[M94] Moszkowski, B.: *Some very compositional temporal properties*, IFIP TC 2 Working Conference on programming concepts, methods and calculi (1994)

[O83] Olderog, E. R.: *On the Notion of Expressiveness and the Rule of Adaptation*, Theoretical Computer Science **24** (1983) 337–347

[P88] Pandya, P: *Compositional Verification of Distributed Programs*, Ph. D. thesis, Tata Institute of Fundamental Research, Bombay (1988)

[PJ91] P. Pandya, and M. Joseph: *P-A logic – a compositional proof system for distributed programs* Distributed Computing **5** (1991)

[R90] Ramesh, S.: *On the Completeness of Modular Proof Systems*, Information Processing Letters **36** (1990) 195–201

[S92] Spivey, Mike: *The Z notation: A reference manual*, Prentice-Hall (1992)

[Q92] Xu Qiwen: *A theory of state-based parallel programming*, Oxford (1992)

[Z89] Zwiers, J.: *Compositionality, Concurrency and Partial Correctness*, Lecture Notes in Computer Science **321** (1989)

[ZBR83] Zwiers, J., de Bruin, A., and de Roever, W. -P.: *A proof system for partial correctness of Dynamic Networks of Processes*, Lecture Notes in Computer Science **164** (1984)

[ZRE84] Zwiers, J., de Roever W. -P., and van Emde Boas, P.: *Compositionality and concurrent networks: soundness and completeness of a proof system.* TR **57**, Nijmegen (1984)

[ZRE85] Zwiers, J., de Roever, W. -P., and van Emde Boas, P.: *Compositionality and concurrent networks: soundness and completeness of a proof system*, Lecture Notes in Computer Science **194** (1985)