Synthesizing different development paradigms:

Combining top-down with bottom-up reasoning about distributed systems[1]

J. Zwiers[2], U. Hannemann, Y. Lakhneche & W.-P. de Roever [3]

Abstract. Our goal is the presentation of a uniform framework for compositional reasoning about the development of distributed processes and data structures.This framework should be a synthesis because, depending on the structure of the processes involved and the verification steps required, *different* formalisms are most suitable for carrying out one's reasoning. We illustrate this uniform framework by presenting a methodology for reasoning about refinement of distributed data structures, i.e., data structures implemented by means of distributed networks. Our synthesis is compositional, state-based, history-based, and contains **sat** style, Hoare style , trace-invariant reasoning and assumption/commitment style specifications as dialects. The resulting formalism can be unfolded as if it were a portable telescope, yielding the style required according to its degree of unfolding.

1 Introduction

Many useful formal techniques for program development are well known by now, and are more and more accepted by software developers. For instance, one sees specifications in Z [S92] or VDM [J86] style, Hoare style pre/postconditions for verification of sequential programs, various logics and proof methods for concurrent programs, data refinement, etc. In practice, various techniques must be *combined*, as it is usually not the case that one single technique covers all aspects, and all stages, of development. There seems to be in particular a substantial gap between *sequential programming* on the one hand, and *concurrency* on the other. Our first contribution is that we show how to incorporate concurrency in the development of systems that start out as purely sequential programs. We are faced here with a so called *specification adaptation problem:* how to switch between a VDM or Z style specification and a trace based specification.

Specification adaptation does not only arise when stepping from sequential to concurrent programming. It also arises within a top-down development of some (sequential or concurrent) system S when one would like to reuse some available component C [H71]. Then C's specification may not immediately fit the required

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

[2] Twente University, P.O. Box 217, 7500 AE Enschede, The Netherlands, zwiers@cs.utwente.nl

[3] Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Preußerstraße 1-9, 24105 Kiel, Germany, {uha,yl,wpr}@informatik.uni-kiel.d400.de

"top-down"specification, and needs to be adapted to this new specification -this reflects the particular interface of C within S. Specification adaptation now consists of proving that C's original specification, which we assume to be given once and for all (e.g., a manufacturer's description of a chip interface), implies this new top-down specification. In case of a relational framework this merely boils down to proving inclusion (or implication, in case of a logical framework). However in case of Hoare-style specification formalisms for sequential systems, such as Z or VDM, and their more complicated versions for concurrent and distributed systems, in which trace-invariants and assumption/commitment pairs are used to characterize the communication interface, the problem becomes how to formulate and *prove* that one specification "implies" another one (in a relatively complete manner). A second contribution is that we show how this can be done; this is a generalization to concurrency of Olderog's masterful "*Note on the rule of adaptation*" [O83], which describes specification adaptation of sequential programs.
The context of these two contributions is our third contribution: a *uniform* framework for *compositonal* reasoning about the development of distributed processes and data structures. Why *uniform*? As already remarked, the choice of the best formalism for carrying out one's formal reasoning depends, among others, on the structure of the eventual implementation and the nature of the particular development step made. For instance, when sequential operations are emphasized, as in the field of data structure implementations, Hoare logics are preferable. Yet when given ready-made modules with given specifications have to be adapted to a given application, i.e., top-down reasoning meets bottom-up reasoning, a **sat** style *relational* formalism is better suited. Consequently a synthesis is called for in which both Hoare style and **sat** style reasoning can be represented [ZdeR 89, HHS87, vKH95].
And why should this framework be *compositional*? Because, following Dijkstra, distributed networks are derived from specifications in a hierarchically structured step-wise process, in which pure specification-oriented terms are gradually replaced by more implementation-oriented structures, as formalized in the so called *mixed term* approach [Wirth 71, Olderog 86, Z89]. In such a mixed term approach the distinction between specifications and programs becomes blurred because program operators may be applied to specifications, and specification operations to programs (e.g., the conjunction of programs $P_0 \wedge P_1$, or the sequential composition of specifications $S_0 ; S_1$). Now if this set-up is going to work, refining a (sub)specification S by one mixed term satisfying S or by another one, which equally satisfies S, should not cause any difference w.r.t. satisfaction of the original top-level specification one started out with. This implies that S should characterize the *complete interface* of that mixed term (incl. all of its refinements) with its environment in the context of that particular refinement step. This calls for a *compositional* framework.
We illustrate this uniform framework by presenting a methodology for reasoning about refinement of distributed data structures, i.e., data structures implemented by means of distributed networks. This imposes the following *requirements* to our

framework: it should incorporate the mixed term approach, easily deal with real life state-based implementations, concurrency, and, as said before, adaptation. Consequently, our synthesis should be, respectively, compositional, state-based, trace-based (to deal compositionally with parallelism), and contain both **sat** style and Hoare style reasoning as dialects.
But this list doesn't cover all desiderata. As mentioned before, a promising way to specify and verify distributed algorithms compositionally in given programming languages for parallel computing is through the use of *assumption/commitment* based [MC81, PJ91] or trace-invariant based [CH81, Z89] formalisms and these methods should also find their place within our approach. The solution to all these at first sight conflicting requirements is a uniform formalism which is a synthesis of all of the above, specialized styles. This synthesis, rather than leading to a system whose complexity is the sum of its subsystems, forces us to adapt the *simplest* formalism available, that of (second order) predicate logic to our purposes by simulating the different styles of reasoning required for each development step within this single framework.
After this introduction we sketch our second order predicate logic formalism and explain how to embed sequential specification styles and CSP structures in it. In section 3 we discuss the relation between trace-based reasoning and state-based reasoning and how to integrate these approaches within a **sat** style formalism or Hoare logics. An approach to a formal solution of the adaptation problem within extensions of Hoare logics for reactive systems is presented in section 4. In order to illustrate these contributions we explain our techniques on a shortest path algorithm and its underlying data structure which is implemented as a network of processes.

2 A uniform framework

We introduce a general logical framework underlying the specification formalisms in this paper. We show how to embed Hoare logic, trace invariant logic, assumption/commitment and **sat** style logics. Thereafter we consider the adaptation problem for these logics. It is shown that adaptation can be done via translation, back and forth, between various logics.

Predicate formulae

We assume as given sets of typed first order variables $x \in Var$, and of typed second order relation variables $X \in VAR$. The exact typing scheme is not important for our goal. Expressions e are built up as usual. Relation constants R and relation variables X are interpreted as $n-ary$ typed relations. 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 ϕ is defined by:
$\phi ::= e_1 = e_2 \mid R(e_1, \ldots, e_n) \mid X(e_1, \ldots, e_n) \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \forall x.(\phi) \mid \forall X.(\phi)$
Formulae like $\phi_0 \rightarrow \phi_1$ are seen as standard abbreviations.

The satisfaction relation

In the sequel we show how, e.g., VDM, Z or Hoare style formulae for CSP processes can be embedded within predicate logic. This requires some notation. All these styles have in common 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 called *base variables* of the specifications ϕ and ψ, expressed by $base(\phi, \psi)$. They are the subset of the free variables of ϕ, ψ consisting of those variables that actually refer to the *observable behaviour* of ϕ and ψ, and correspond in general to program variable, trace variables, ready-set variables, termination flags, and the like [P88]. The remaining first order variables 0f ϕ, i.e. the non-base ones, are denoted by $lvar(\phi)$, and are called the *logical variables* of ϕ, in the literature also referred to as "freeze" or "rigid" variables.

Sequential systems

A *relational specification* is a predicate formula with occurrences of unprimed and primed versions of variable names. In the examples we use a VDM style notation of the form:

require $pre(x_0, \ldots, x_n)$
ensure $post(x_0, \ldots, x_n, x_0{\prime}, \ldots, x_n{\prime})$

which, for our purpose, abbreviates the predicate:

$pre(x_0, \ldots, x_n) \;\Rightarrow\; post(x_0, \ldots, x_n, x_0{\prime}, \ldots, x_n{\prime})$

A *Hoare specification* is of the form $\{pre(\bar{x}, \bar{v})\}\ S\ \{post(\bar{x}, \bar{v})\}$, where we have pre/postconditions with free program variables $\bar{x}$ and logical variables $\bar{v}$. This formula can now be seen as an abbreviation that S satisfies the relational predicate $\forall \bar{v}.(pre(\bar{x}, \bar{v}) \Rightarrow post(\bar{x}{\prime}, \bar{v}))$. A convenient adaptation rule for turning relational specifications into Hoare style pre-postconditions is based on weakest preconditions or strongest postconditions defined for relational predicates. Let $\psi(\bar{x}, \bar{x}{\prime})$ be a relational predicate, and let $pre(\bar{x}, \bar{v}), post(\bar{x}, \bar{v})$ be a state predicates, with possible occurrences of logical (i.e. non-state) variables $\bar{v}$. We define:

$$wp(\psi, post) \stackrel{\text{def}}{=} \forall \bar{x}{\prime}.(\psi \Rightarrow post[\bar{x}{\prime}/\bar{x}])$$
$$sp(pre, \psi) \stackrel{\text{def}}{=} \exists \bar{x}{\prime\prime}.(pre[\bar{x}{\prime\prime}/\bar{x}] \wedge \psi[\bar{x}{\prime\prime}/\bar{x}, \bar{x}/\bar{x}{\prime}])$$

The adaptation of Hoare formulae to relational ones is given above. Conversely, if a program S has been specified by a relational predicate ψ, then for any desired postcondition $post$, we know that S satisfies the formulae $\{wp(\psi, post)\}\ S\ \{post\}$.

Trace logic and CSP

As an application of the formalism above we recall how to deal with trace based specifications for CSP style processes [Hoare 85]. Trace based specifications are assertions of the form $S(h)$ with free occurrences of a trace typed variable h, denoting the communication history of a process. A *trace* is a finite sequence of communications taken from a given set of communication actions $Comm$. We employ standard notations and operations for trace expressions t and finite alphabets $\alpha \subseteq Comm$. We assume that α contains the special symbol "$\surd$", which by convention is used to signal termination. Furthermore, we need the following trace operations:

ε	the empty trace, also written as $<>$
$< a >$	a trace with a single communication $a \in \alpha$
$t_0 \frown t_1$	concatenation of t_0 and t_1
$t \upharpoonright \alpha$	projection onto α
$t[b/a]$	renaming occurrences of a into b within t
$last(t)$	the last communication of t
$rest(t)$	t except for $last(t)$
$fin(t)$	abbreviates $last(t) = \surd$
$t_0; t_1 = t_0$	if $\neg fin(t_0)$,
$t_0; t_1 = rest(t_0) \frown t_1$	if $fin(t_0) \wedge t_1 \neq <>$
$t_0; <>= t_0$	
$t_0 \preceq t_1$	t_0 is a *prefix* of t_1

Note that when $t_1 \neq <>$, no $\surd$-symbol occurs *within* $t_0; t_1$, even if t_0 originally ended with a $\surd$. Our basic programming language consists of atomic communications, parallel and sequential composition of processes, choice of processes, hiding and renaming of channels:

$P ::= stop \mid skip \mid a \mid P_0 \,;\, P_1 \mid\ P_0 \| P_1 \mid P_0 \sqcap P_1 \mid P \backslash c \mid P[d/c]$

It is known from the literature [CH81, Z89] how to define for a given CSP process P its *alphabet* $\alpha(P)$ and an equivalent trace specification $S(h)$. We give a brief sketch of this.

$stop(h)$	$h = \varepsilon$
$skip(h)$	$h = \varepsilon \vee h =< \surd >$
a	$h \preceq < a > \frown < \surd >$
$P_0 \,;\, P_1$	$\exists t_0 \exists t_1 . P_0(t_0) \wedge P_1(t_1) \wedge h = t_0; t_1$
$P_0 \| P_1$	$P_0(h \upharpoonright \alpha(P_0)) \wedge P_1(h \upharpoonright \alpha(P_1)) \wedge h \upharpoonright \alpha = h$, where $\alpha = \alpha(P_0) \cup \alpha(P_1)$
$P_0 \sqcap P_1$	$(P_0(h) \wedge h \upharpoonright \alpha - \alpha(P_0) = \varepsilon) \vee (P_1(h) \wedge h \upharpoonright \alpha - \alpha(P_1) = \varepsilon)$ where $\alpha = \alpha(P_0) \cup \alpha(P_1)$
$P \backslash c$	$\exists t . P(t) \wedge h = t \upharpoonright (\alpha(P) - \{c\})$
$P[d/c]$	$\exists t . P(t) \wedge (h = (t \upharpoonright \alpha(P))[d/c]$

Thus we may regard CSP processes as abbreviations for certain trace specifications within our framework. The special variable h is the only base variable of such process specifications, i.e. all other free variables are logical variables. This implies that a relation of the form "$P\,\mathbf{sat}\,Q$" must be read as "$\forall h . (P(h) \;\rightarrow\; Q(h))$". Formally speaking, both P and Q are trace formulae. However, it useful to distinguish the cases where P or Q are actually written down as CSP processes. Formulae that do not contain CSP operators at all will be called (logical) specifications.

3 Integrating trace- and state- based reasoning

How to incorporate concurrency into existing methods for development of sequential systems? It is well known that within process languages like CSP, *data* can be represented by *processes* with appropriately named channels for the operations that are allowed on the data. One can use a trace logic based on the **sat** relation to specify and verify such data processes. The problem then is: how to incorporate these results into a *sequential* program context? One might switch here completely to a formalism for concurrency, since data can now be seen as processes. However, it is advantagous to work at a more abstract level, relying on so called *virtual states*. The solution for this problem is related to techniques for data refinement. The step from traces to states is made formal by means of *retrieve functions* similar to those used in [J86]. Occasionally, one also needs *retrieve relations*, like those used in [Reynolds 81]. The basic idea is to introduce a process T for each abstract datatype T. The operations $op_1, \ldots, op_n$ defined for T data are taken as the names of channels of this T process. A declaration of the form: **var** x:T; S is then refined into (x.T // S)$\backslash x.op_0, \ldots, x.op_n$. As usual, x.T is the process T with the channels $op_1, \ldots, op_n$ renamed into $x.op_1, \ldots, x.op_n$. Also, x.T // S denotes parallel composition where x.T is *subordinate* to S, in the sense of [Hoare 85]. That is, termination is determined by process S, and process x.T stops as soon as process S terminates.

What should be the specification for the T process? We derive it from the specification of the abstract datatype. The main stepping stone here is the definition of a retrieve relation retr_T that maps traces of a T process to states of an abstract T object. In general we assume that the T operations can be classified as either *constructor operations* $cons_i : T \times T_1 \times \cdots \times T_n \;\rightarrow T$, for $i \in I$, or as *selector operations* $sel_j : T \times T_1 \times \cdots \times T_m \;\rightarrow T'$, for $j \in J$. (The types T_i and T' are assumed to be different from T.)
The operations are specified by relational predicates of the form: $\psi_i(a, x_1, \ldots, x_n, a\prime)$ (for constructors) and $\phi_j(a, x_1, \ldots, x_m, y)$ (for selectors). We define, in a canonical fashion, a retrieve relation $retr\_T(h, a)$. Informally, $retr\_T(h, a)$ holds iff a is one of the possible abstract values for T which result from applying the list of operations recorded in trace h. Assume that objects of type T have an initial value val_{init}. $retr_T$ is determined by the following:

$retr\_T(\langle\,\rangle, val_{init}) \;\stackrel{\text{def}}{=} \textbf{true},$
$retr\_T(h^\frown \langle(cons_i, v_1, \ldots, v_n)\rangle, a) \stackrel{\text{def}}{=} \exists\tilde{a}.(retr\_T(h, \tilde{a}) \wedge \psi_i(\tilde{a}, v_1, \ldots, v_n, a)),$
$retr\_T(h^\frown \langle(sel_j, v_1, \ldots, v_m)\rangle, a) \stackrel{\text{def}}{=} retr\_T(h, a).$

The next step is to define a trace specification $\chi_T(h)$ for the T process, based upon this retrieve function. Informally, it states that whenever selector operation sel_j is applied, the value returned is defined by ϕ_j, where the current *state* a is determined by $retr\_T(h, a)$:

$\chi_T(h) \stackrel{\text{def}}{=} \forall v_1, \ldots, v_m, y.(\forall t \leq h. \bigwedge_j(last(t) = (sel_j, v_1, \ldots, v_m, y) \;\Rightarrow$
$\exists\tilde{a}.(retr\_T(rest(t), \tilde{a}) \wedge \phi_j(\tilde{a}, v_1, \ldots, v_m, y))))$

While this is a purely trace oriented style we preferably use formalisms where we can refer to data objects explicitly in our assertions. Thus, we incorporate these techniques into a state-trace formalism which in fact denotes an abbreviation of pure trace expressions. Let $p(h, x)$ be a predicate formula with free occurrences of the trace h and of variable x of type T. We define the meaning of such pred-

icates by a translation back to (pure) trace formulae: $p(h,x)$ abbreviates $\tilde{p}(h)$, defined by: $\tilde{p}(h) \stackrel{\text{def}}{=} \exists a : T.\,(retr\_x.T(h,a) \wedge p(h,a))$, where
$retr\_x.T(\langle\,\rangle, val_{init}) \stackrel{\text{def}}{=} \mathbf{true}$,
$retr\_x.T(h^\frown\langle(x.cons_i, v_1, \ldots, v_n)\rangle, a) \stackrel{\text{def}}{=} \exists\tilde{a}.(retr\_x.T(h,\tilde{a}) \wedge \psi_i(\tilde{a}, v_1, \ldots, v_n, a))$,
$retr\_x.T(h^\frown\langle(x.sel_j, v_1, \ldots, v_m)\rangle, a) \stackrel{\text{def}}{=} retr\_x.T(h,a)$, the retrieve function for variable x of type T. In the sequel we will regard all predicates as predicates on traces and such state variables unless stated otherwise.

Specification adaptation and adaptation completeness

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, iteration, etc. Since they combine programming constructs with logic 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 holds. When T actually has the form of a logic specification and R the form of a program, then this describes a classical, top-down development step, an *implementation relation*. When both T and R are programs, we have a classical program transformation step, a *refinement relation*. Finally, when both T and R are logic specifications, we are dealing with *specification adaptation*. Such an adaptation step in the development of an implementation is in general necessary when one would like to implement T by some already available module M, where M is known to satisfy some given specification R. We do not want to verify *directly* that M **sat** T, since this would force us to inspect the internal structure of M. In fact, our programming language might include encapsulation constructs that do not even allow us to inspect this internal structure! And,e.g., chip manufacturers often produce manuals to describe the structure of their product; here such manuals fulfill the rôle of R. Therefore, we have to check indirectly that M **sat** T, by showing that R **sat** T, since M **sat**R is given once-and-for-all. 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 (as illustrated in section 5).

In order to close this gap between two specifications one may apply rules that do not refer to the internal structure of their process objects (programs or formulae), but express the closure properties of the domain these objects belong to. In the terminology of Zwiers [Z89], a **sat** proof system is *adaptation complete* if, whenever for specifications P, Q,

$\models\ \forall X.(\,(X\ \mathbf{sat}\ P) \rightarrow (X\ \mathbf{sat}\ Q)\,)$

holds, then this implication is provable within this proof system. This is a simple task for a purely predicate based **sat** style, for it is clear that $\models \forall X.((X\ \mathbf{sat}\ P) \rightarrow (X\ \mathbf{sat}\ Q))$ holds iff $\models\ P\ \mathbf{sat}\ Q$, which for a **sat** style system is a simple verification condition. However in Hoare, trace-invariant based and assumption/commitment style formalisms this becomes a problem which is solved in the sections below.

Transformations between specification styles

Simple proof systems have been built for the **sat** relation for CSP [H69, Z89, Olderog 86]. Specification adaptation is rather straightforward here, as argued above. This is certainly one of the advantages of such **sat** systems. Yet, **sat** based logics do not treat sequential composition very well. For instance, consider the following proof rule for **sat** logic:

Rule 1 (Sequential Composition Sat style)

$$\frac{S_0 \,\mathbf{sat}\, Q_0\ ,\ S_1 \,\mathbf{sat}\, Q_1}{S_0; S_1 \ \mathbf{sat}\ Q_0; Q_1}$$

Note that when Q_0 and Q_1 are logical specifications, then $Q_0; Q_1$ still contains a sequential operator that can be eliminated only at the expense of introducing an existential quantifier, i.e. in this style a proper rule for sequential composition doesn't eliminate this operator but reduces it to a new one at the level of specifications, as e.g., observed by Lamport [L91]. A nicer rule can be obtained by embedding Hoare logic within the **sat** system. For any trace predicate $S(h)$, define a *relation* on traces "$;S$" as follows:

$(;S)(h, h') \stackrel{\text{def}}{=} \exists h''.(S(h'') \wedge h' = h; h'')$

Now Hoare logic is embedded in the **sat** system by $\{P\}\,;S\,\{Q\} \stackrel{\text{def}}{=} P; S\,\mathbf{sat}\, Q$

Note that the ";" on the right hand side of this definition is the operator defined in section 2. The main advantage of this approach is that now the following simple composition rule holds for these **sat** style formulae in Hoare style disguise:

Rule 2 (Sequential Composition Hoare style)

$$\frac{\{P\}\,;S_0\,\{R\}, \{R\}\,;S_1\,\{Q\}}{\{P\}\,;(S_1; S_2)\,\{Q\}}$$

Similar Hoare style rules for these disguised **sat** style formulae can be given for other sequential constructs, such as the rule for iteration based on loop invariants. Dealing with relations on traces requires similar conventions on the use of variables as in section 2. We use h as trace variable in preconditions and h' for the free trace variable in postconditions. A novel aspect is that within $\{P\}\,;S\,\{Q\}$ the pre and postconditions P and Q are *trace* specifications and may be CSP processes themselves. Transformation rules between **sat** specifications and Hoare style formulae are:

Rule 3 (Hoare- sat)

$$\frac{\forall \bar{u}.(\{P\}\,;S\,\{Q\})}{S\ \mathbf{sat}\ \forall \bar{u} \exists t.(P(t) \rightarrow Q)}$$

Rule 4 (sat - Hoare)

$$\frac{S\ \mathbf{sat}\ Q}{\{wp(Q, R)\}\,;S\,\{R\}} \qquad\qquad \frac{S\ \mathbf{sat}\ Q}{\{R\}\,;S\,\{sp(R, Q)\}}$$

Here Q' abbreviates $Q[h'/h]$, $\bar{u}$ expresses the set of logical variables of P and Q, and $wp((;S),Q)$ and $sp(P,(;S))$ are trace logic equivalents to the definitions from section 2 (for variables $\overline{x}$).

As for classical Hoare formulae, one can define adaptation rules here too, as in , e.g., [O83, D92] for sequential constructs and [Z89] for distributed ones. Yet it appears to be simpler to translate back and forth between Hoare formulae and "**sat**" style formulae, due to the fact that adaptation is so simple to express in **sat** style systems. After all Hoare style formulae can be seen as mere abbreviations of "**sat**" formulae, and conversely, if S **sat** R, then S satisfies the Hoare formula $\{P\}\,;S\,\{P;R\}$.

4 Specification adaptation for extensions of the Hoare style system for CSP processes

For practical verification of reactive distributed systems, 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 a control point immediately after process S, whereas traces without such a "tick" correspond to intermediate stages of the execution of process S. In [ZRE85] a class of formulae was introduced where pre/postconditions P and Q describe only traces ending with a "$\surd$", corresponding to snapshots before and after execution of S, and where the specification of traces without a "$\surd$" is delegated to a *trace invariant* I which refers to the communication history only, but not to state variables:

$$I\ :\ \{P\}\ S\ \{Q\}\ \stackrel{\text{def}}{=}\ \{I \wedge (fin \rightarrow P)\}\ ;S\ \{I \wedge (fin \rightarrow Q)\}$$

The formula above is valid, whenever: if **(i)** the computation starts in a terminated state such that P holds, then **(ii)** at every moment of execution of S the invariant I holds, and **(iii)** Q holds in case, and when, this execution terminates. This specification style separates local conditions such as pre- and postcondition upon the initial and final states from the communication interface w.r.t. the entire system, without losing the simplicity of the rules of Hoare logic which are slightly modified to fit this specification style.

A further extension of the invariant style splits up the invariant in two parts by distinguishing between the communication events of the environment and the communication events of the specified process. This assumption / commitment style was introduced by Misra and Chandy [MC81] in order to find a practically convenient formalism for distributed processes. The assumption refers to the expected communication behaviour of the environment and the commitment refers to the communications of the specified module, provided that the assumption has not been violated before by the environment. The attraction of this style (and of the related rely/guarantee formalism of [J81]) is that it enables the formulation of a proof rule for parallelism which is closest to Hoare's invariant rule for iteration in that both incorporate an induction argument on the length of the computation of which the soundness is established *once and for all* in their soundness proofs, and which, therefore, needn't be justified upon every application of these rules, as it is the case with the above trace invariant formalism. Proof rules for a state-trace based model were given in [ZdeBdeR 83] as derivations of rules of the invariant system. A relatively complete compositional proof system in this assumption/commitment style has been given in [P88], called *P-A Logic*.

Let A, C be trace assertions that do not refer to state variables. We define $\bullet A$ and $\mathcal{K}ern(A)$ by

$\bullet A(h) \stackrel{\text{def}}{=} (h \neq \langle\rangle \rightarrow A(rest(h)))$,

$\mathcal{K}ern(A)(h) \stackrel{\text{def}}{=} \forall h'(h' \leq h.(A(h')))$, intuitively denoting the largest prefix closed subset of A. An assumption/commitment specification is then defined by

$(A, C) : \{P\}\ S\ \{Q\} \stackrel{\text{def}}{=} \mathcal{K}ern(\mathcal{K}ern(\bullet A) \rightarrow C) : \{P\}\ S\ \{\mathcal{K}ern(A) \rightarrow Q\}$

and can therefore be regarded as a special case of the trace invariant formalism. Note that $\mathcal{K}ern(\mathcal{K}ern(\bullet A) \rightarrow C)$ expresses a trace invariant because of the outermost occurrence of the prefix closure operator $\mathcal{K}ern$.The general idea behind this style is that this correctness formula holds, whenever if **(i)** P holds for the initial terminated state and trace, then **(ii)** commitment C holds initially and after every communication provided A holds after all preceeding communications and **(iii)** if S terminates and A holds after all communications, then Q holds for the final state. Observe that $\mathcal{K}ern(\bullet A) \rightarrow C)$ is a literal translation of **(ii)** and $\mathcal{K}ern(A) \rightarrow Q$ a literal translation of **(iii)**, where $A(<>)$ holds by definition. While again the proof rules of Hoare logics have to be adapted slightly, an interesting rule for parallel composition is applicable: if we have specifications of the form $(A_i, C_i) : \{P_i\}\ S_i\ \{Q_i\}$, for $i = 0, 1$ such that for some assertion A, $(A \wedge C_i) \rightarrow A_{i-1}$ for $i = 0, 1$ hold, then the rule yields a specification for $S_0 \| S_1$ of the form:

$(A, C_0 \wedge C_1) : \{P_0 \wedge P_1\}\ S_0 \| S_1\ \{Q_0 \wedge Q_1\}$.

As remarked before, the advantage of this rule is that it incorporates an induction argument on the length of the communication trace which is proved sound once and for all in the soundness proof of this rule, and needn't be repeated upon every application, as demonstrated in [ZdeBdeR 83].

Solving the adaptation problem for the trace invariant formalism

As a result of the different approaches towards the design of a process we may have a gap between the specification of the already developed module and the requirements as given by the proof obligations of the application process. This adaptation problems turns out to be more complicated: assuming that $\models I_1 : \{p_1\}X\{q_1\} \rightarrow I_2 : \{p_2\}X\{q_2\}$ holds, we have to prove formally that $I_2 : \{p_2\}X\{q_2\}$ is derivable from $I_1 : \{p_1\}X\{q_1\}$. Clearly this proof is more difficult than the one for the **sat** style and requires some more subtile rules than just an application of the consequence rule. Within this distributed set-up, the semantics of programs (or specifications) involve more than state-based logic, in particular specific closure properties which are a consequence of the trace-based approach in the present representation, e.g., a program P is prefix-closed, hence $P = \mathcal{K}ern(P)$. Concerning the invariant formalism we sketch two different ways to include this additional information into the formal system. By expanding the definitions and applying some logical simplifications, one can show how to transform these formulae into equivalent **sat** style formulae, i.e., that $I : \{\ P\ \}\ X\ \{\ Q\ \}$ is equivalent to X **sat** $I : P \rightsquigarrow Q$, where

$(I : P \rightsquigarrow Q)(h) \ \stackrel{\text{def}}{=}\ \forall t. fin(t) \wedge P(t) \rightarrow (I(t;h) \wedge (fin(t;h) \rightarrow Q(t;h))\,)$

The adaptation problem for trace invariant formulae can thus be reduced to that between **sat** style formulae and is therefore in principle solved. This approach introduces a certain amount of complexity due to the transformations involved. A more promising way is therefore to adapt the specification directly within the invariant proof system itself. For a given precondition R and characterization $(I : P \rightsquigarrow Q)$ of a given black box module X both the *strongest invariant* and the strongest postcondition are formulated:

$sinv(R, (I : P \rightsquigarrow Q)) \ \stackrel{\text{def}}{=}\ R; (\forall t.(P(t) \rightarrow I(t;h))$

$sp(R, (I : P \rightsquigarrow Q))\,) \ \stackrel{\text{def}}{=}\ R; (\forall t.(P(t) \rightarrow I(t;h) \wedge Q(t;h)))$.

Using these constructs, an intuitivly appealing adaptation rule can be added to the proof system:

Rule 5 (Adaptation rule)

$$\frac{I : \{\ P\ \}\ S\ \{\ Q\ \}}{sinv(P_1, (I : P \rightsquigarrow Q)) : \{\ P_1\ \}\ S\ \{\ sp(P_1, (I : P \rightsquigarrow Q))\ \}}$$

Since $\models (I : P \rightsquigarrow Q) \ \rightarrow (I_1 : P_1 \rightsquigarrow Q_1)$ holds, we can conclude that $\models sinv(P_1, (I : P \rightsquigarrow Q)) \rightarrow I_1$ and $\models sp(P_1, (I : P \rightsquigarrow Q)) \rightarrow Q_1$ hold. Adaptation completeness of the enlarged invariant style proof system now follows immediately. This rule is related to the adaptation rules for sequential Hoare style systems that have been discussed by Olderog [O83].

A similar rule can be formulated for the assumption/commitment style by transforming a formula into its invariant style equivalent, applying the rule above and lifting the resulting formula up to its original formalism: Abbreviate the invariant style equivalent of $(A, C) : \{P\}\ S\ \{Q\}$ by $(\,(A, C) : P \rightsquigarrow Q)$. Within the invariant style we can apply the adaptation rule 5 to obtain

$sinv(R, (\,(A, C) : P \rightsquigarrow Q)) : \{\ R\ \}\ S\ \{\ sp(R, ((A, C) : P \rightsquigarrow Q))\ \} \qquad (\dagger)$

This formula can be interpreted both as an invariant formula and as an assumption/commitment formula, since every invariant formula $I : \{P\}S\{Q\}$ trivially denotes the assumption/commitment formula $(\mathbf{true}, I) : \{P\}S\{Q\}$. In this derivation of an adaptation rule we introduce a new assumption A' in a straightforward way: If the formula $(\dagger)$ holds, we conclude that also for every A' the assumption/commitment formula

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

holds (Note that A' on the assumption position corresponds to $\mathcal{K}ern(\bullet A)$ at the commitment position). Thus, one can formulate a rather complex adaptation rule within the assumption/commitment formalism itself:

Rule 6 (Adaptation Rule)

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

The adaptation completeness proof of an assumption/commitment proof system including this rule is completely analogous to that for the trace invariant system, using a rule of consequence adapted to the assumption/commitment framework [Ha94]. Thus we are able to prove all valid implications between specifications of black box modules within this special framework as well as within the other formal systems discussed in this paper, be it the trace invariant framework, Hoare logic or the **sat** style proof system.

5 Integration of techniques

Our various adaptation problems are illustrated by developing a shortest path algorithm. This algorithm makes essential use of a priority queue. During the sequential part of its development, an a priori given state-based specification of this priority queue is used, illustrating the kind of specification adaptation required in case of modular reasoning, i.e., when a "clash" occurs between top-down and bottom-up development. When implementing this priority queue by a distributed network whose compositional trace-based specification needs to be adapted to show that it satisfies the state-based specification, we take again advantage of our uniform framework and present a canonical solution for these kind of problems.

Example - sequential part

Our example starts out with an abstract algorithm for calculating shortest paths from a given start node x, originally given in [D59]. Since we are aiming at *modular* development ,i.e. based on a priori given progam modules, we use a programming notation that is in the style of object oriented languages like Eiffel [M88]. We represent a directed graph by a set Node, where each Node y has a some attributes: "y.name" is a unique node identifier, and attribute "y.edges" is a set of pairs (z,d) where z is a Node and d is a real number. Informally, when (z,d)$\in$ y.edges, then there is an edge from y to z, with distance d. By "reachable(x,y)" we denote the predicate:

$\exists x_0, d_1, x_1, \ldots, d_n, x_n.(x = x_0 \wedge y = x_n \wedge \bigwedge_{i=1,n}(x_i, d_i) \in x_{i-1}.edges)$

In a similar style, one defines the partial function "distance(x,y)", denoting the distance of a shortest path from x to y, provided that reachable(x,y) is true. The algorithm assumes two more Node attributes: "y.reached" is a boolean indicating whether a path to y has been found. If y.reached is true, then "y.distance" contains the distance of some path from x to y. The procedure "path" itself is specified by a Hoare style pre-postcondition pair.

procedure path(x:Node)
pre $\forall$y:Node.($\neg$y.reached) $\wedge$ $\forall$(z,d) $\in$ y.edge. d $\geq$ 0
post $\forall$y:Node. (y.reached $\iff$ reachable(x,y)) $\wedge$
(y.reached $\Rightarrow$ y.distance = distance(x,y))

The algorithm itself, although not long, is quite intricate. A Hoare style correctness proof can be found in [Reynolds 81]. Rather than redoing that proof, we would like to focus on some relevant aspects for *modularity*, i.e. reasoning on the basis of program modules. The algorithm, as shown below, uses a so called *priority queue variable*, named "unexplored". Nodes y in "unexplored" have been found to be reachable, but the value of y.distance does not need to be the *shortest* distance from x to y as yet. *New* paths are found by extending those paths that end in some "unexplored" node. The correctness of the algorithm below crucially depends on the *order* of processing nodes from "unexplored" ones: nodes y are processed in the order of their y.distance attribute. The procedure "getmin" yields an element that is *minimal* among the nodes currently in the priority queue and deletes that element.

```
    procedure path(x:Node)
var   unexplored: Prioqueue; y,z: Node; d: Real
begin x.reached := true; x.distance:=0; unexplored.insert(x,0) ;
        while not unexplored.isempty do
                y:=unexplored.getmin;
                for (z,d) in y.edges do
                        if not z.reached then z.reached := true;
                                z.distance := y.distance + d; unexplored.insert(z);
                        elsif y.distance + d < z.distance then
                                z.distance := y.distance + d; unexplored.insert(z);
                        end ;
                end ;
        end ;
end path
```

The correctness proof of procedure path in [Reynolds 81] is based on a number of state invariants, for instance:

$$\text{geninv} \stackrel{\text{def}}{=} (\forall \text{v,w:Node.}(\text{v.reached} \wedge \text{v} \notin \text{unexplored} \wedge \text{w.reached} \wedge \text{w} \in \text{unexplored}) \Rightarrow \text{v.distance} \leq \text{w.distance})$$

In order to check invariance, one should consider each program step in turn, but for now, let us focus on the "unexplored.getmin" step. Reynolds' proof includes the following Hoare formula:

$$\{\text{geninv} \wedge \text{unexplored} \neq \emptyset\}\ \text{y:=unexplored.getmin}\ \{\text{geninv}\} \qquad (*)$$

(Similar formulae are present for the steps of the form "unexplored.insert(z)".)

Now, let us assume that we do have an already existing "Priority queue" module. After a simple syntactic renaming, such that it actually operates on "Nodes", we have a specification as listed below.

class Prioqueue
export insert, getmin, isempty; **var** queue: Set[Node]

procedure insert(x: T); **ensure** $\text{queue}' = \text{queue} \uplus \{\text{x}\}$

function getmin: T
require $\text{queue} \neq \emptyset$
ensure $\text{queue} = \text{queue}' \uplus \{\text{getmin}\} \ \wedge \forall \text{x} \in \text{queue.}\ (\text{getmin.distance} \leq \text{x.distance})$

function isempty: Bool; **ensure** isempty $\Leftrightarrow$ (queue = $\emptyset$)

end Prioqueue

In essence, this is a specification in VDM or Z style, so there is a gap between the Hoare formula ($*$) for "getmin" above, and the specification within the Prioqueue class (At the end of this section we show how ($*$) can be derived from this specification). One might remark that we should have specified the Prioqueue class with Hoare formulae, too. For instance, using this formula:

$\forall q.$({queue = q} y:=unexplored.getmin {q = queue $\uplus$ y $\wedge$
$\forall$x $\in$ queue. (y.distance $\leq$ x.distance)})

Even here we have an adaptation problem: the desired specification does not follow from the one above by means of the rule of consequence, because it quantifies over all v s.t. $v \notin unexplored \wedge v.reached$ and not just y.

Now, specification adaptation for *sequential* programs is not difficult, because, as argued above, it can always be reduced to checking logical implication between relational predicate formulae. Z style specifications are by nature relational formulae, **require -ensure** specifications have been reduced to such formulae above, and VDM specifications can be treated analogously.

For example, from the Prioclass specification it follows that the assignment y:=unexplored.getmin satisfies the relational predicate $\psi(queue, queue\prime, y\prime)$, defined by:

queue $\neq \emptyset \Rightarrow$ (queue = queue$\prime \uplus$ {y$\prime$} $\wedge$ $\forall$x $\in$ queue. (y$\prime$.distance$\leq$ x.distance))

Hoare formula ($*$) can now be verified by calculating the weakest precondition with respect to postcondition "geninv", and by checking that this is implied by "geninv $\wedge$ unexplored $\neq \emptyset$"

Example - concurrent part

For our shortest path example, the Prioqueue can play the role of type T. The corresponding Prioqueue *process* could be the following:

```
process Prioqueue chan   insert,getmin: Node; delete: Node; isempty: Bool
begin  do insert?x      → (Buffer(x) \\
                           Prioqueue[down/insert,up/getmin,ise/isempty,
                                     del/delete])\ise,del
       or isempty!true → skip
       or delete?x      → skip
       od
end Prioqueue
```

This process represents the empty queue, which is extended with a buffer process for each inserted element. Its communication interface satisfies the canonically constructed specification $\chi_{Prioqueue} \stackrel{\text{def}}{=}$

$\forall t \leq h.(last(t) = (getmin, x) \Rightarrow \exists v.(retr_{Prioqueue}(t, v) \wedge getmin(v, min(v))$
$\wedge last(t) = (isempty, \mathbf{true}) \Rightarrow retr_{Prioqueue}(t, \emptyset)$

towards its environment, while it internally uses a Prioqueue process itself, and a Buffer process as data processes.

```
process Buffer(x:Node);
chan  insert, getmin, up, down:Node; delete,del: Node; isempty,ise: Bool
begin   empty := false;
```

```
    while not empty do
      if insert?y          → if x.name=y.name then x.distance:=y.distance
                             elsif x.distance<y.distance then down!y
                             else down!x; x:=y fi
      or delete?y          → del!y;
                             if x.name=y.name then ise?empty
                                if not empty then up?x fi fi
      or getmin!x          → ise?empty; if not empty then up?x fi
      or isempty!false     → skip
      fi
    od
end Buffer
```

For the actual verification of this process an invariant specification is more convenient, e.g., $\chi_{Prioqueue}[down/insert, up/getmin] \rightarrow \chi_{Prioqueue}$:

$$\{retr_{Bag} = [x]\}\ Buffer \{retr_{Bag} = \emptyset\}$$

This specification states that whenever the *Buffer*'s internal data structure behaves as a priority queue, its communication to the environment will satisfy the priority queue specification itself.

Now assume that a *Buffer* process did already exist within some library, proven correctly w.r.t. a *different* specification, i.e., we are not given the program text above , but an invariant specification of some black box object *Buffer*. Since it is not possible to verify that library text again against the desired specification, our only possibility is to use the formal adaptation rules to derive the desired specification from the given one. As a result of section 4, we get that whenever one specification implies another one this can be proven within the proof system that was shown to be adaptation complete [Ha94](containing also a worked-out example of specification adaptation in case of a buffer, similar to the one above).

6 Conclusion

Integrating various techniques for software development is highly desirable. The need for integration arises in particular in a *modular* aproach, where existing software modules might have been specified in a different formalism than the one used by the developers of the system. This is even unavoidable when during the development process a switch is made from sequential programming to a parallel implementation. We have shown how to achieve this integration, by the technique of *specification adaptation.* Various forms of adaptation, as well as the issue of adaptation completeness, have been discussed. Some of these techniques have been illustrated by developing a shortest path algorithm. The technique used in this example for switching from sequential to concurrent systems is related to techniques for data refinement, in the style of [J86].

Finally, we remark that many of the techniques that we displayed in this paper could be formulated only because of the presence of an underlying uniform framework that is based on predicate logic.

References

[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.

[D59] Dijkstra,E.W., *A note on two problems in connexion with graphs*, Numerische Mathematik 1, 1959

[Ha94] Hannemann, U., *Modular complete proof systems for distributed processes.*, Kiel, 1994.

[H69] Hoare, C.A.R., *The axiomatic basis of programming*, CACM, 1969.

[H71] Hoare, C.A.R., *Procedures and parameters: An axiomatic approach*, LNM,1971.

[Hoare 85] C.A.R. Hoare: *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HHS87] Hoare, C.A.R., He Jifeng and Sanders,J.W.,*Prespecification in Data Refinement*, IPL 25, 1987.

[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.

[vKH95] B. v.Karger and C.A.R. Hoare:*Sequential Calculus*, IPL 53, 1995

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

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

[MC81] Misra, J. and Chandy, K.M., *Proofs of networks of processes.* IEEE TSE, 7, 1981.

[O83] Olderog, E.R., *On the Notion of Expressiveness and the Rule of Adaptation*, TCS 24, 1983.

[Olderog 86] E.-R. Olderog: *Process theory: Semantics, specification and verification.* LNCS 224, 1986.

[P88] Pandya, P. *Compositional Verification of Distributed Programs*,Bombay, 1988.

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

[Reynolds 81] J.C. Reynolds:*The craft of programming.* Prentice-Hall, 1981.

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

[Wirth 71] N. Wirth: *Program development by stepwise refinement.* Communications of the ACM, Vol. 14, No. 4, pp. 221 – 227, 1971.

[ZdeR 89] J. Zwiers, W.-P. de Roever: *Predicates are Predicate Transformers: a Unified Compositional Theory for Concurrency.* Proceedings of the 8th Symposium on Principles of Distributed Computing, pp. 265 – 279, 1989.

[ZdeBdeR 83] Zwiers,J., de Bruin, A. and de Roever, W.-P., *A proof system for partial correctness of Dynamic Networks of Processes.* , LNCS 164, 1984.

[Z89] Zwiers, J. *Compositionality, Concurrency and Partial Correctness*, LNCS 321, 1989.

[ZRE85] Zwiers, J., de Roever, W.-P. and van Emde Boas,P. *Compositionality and concurrent networks: soundness and completeness of a proof system.* , LNCS 194, 1985.