On Refinement and Temporal Annotations *

Ron van der Meyden[1] and Yoram Moses[2]

[1] School of Computer Science and Engineering
The University of New South Wales, Sydney 2052, Australia
meyden@cse.unsw.edu.au
[2] Department of Electrical Engineering
Technion, Haifa, Israel
moses@ee.technion.ac.il

Abstract. This paper introduces the semantics of a wide spectrum language with a rich compositional structure that is able to represent both temporal specifications and sequential programs. A key feature of the language is the ability to represent partial correctness annotations expressed in temporal logic. A refinement relation is presented that enables refinement steps to make use of these partial correctness assertions. It is argued by means of an example that the approach presented allows for more flexible reasoning using temporal annotations than previous approaches, and that the added flexibility has significant value for program optimization.

Keywords: Refinement calculus, temporal logic, temporal refinement calculi

1 Introduction

Work on program refinement can be categorised into two classes. One of the most deeply explored approaches [Mor90,BvW98,Mor87] is *state-based*, premised on the use of predicate transformers and weakest preconditions as a semantic basis. This is natural for programs whose specifications are descriptions of input/output relations, of which predicate transformers are a generalization. One of the advantages of this approach is the ability to write specifications containing constructs modelling two distinct types of annotation. Annotations of one type, which we call *coercions*, are used to state properties that the program is *required* to satisfy. Annotations of the other type, which we call *assertions*, are more like conventional program annotations in that they are used to state properties that the program has been *proved* to satisfy.

Another class of refinement calculi, motivated directly by distributed programs, is *action-based* [GS86,Hol89,HL95,Win86,Lam94]. Work in this category typically begins with a process calculus. In a distributed setting, specifications and reasoning very

* To appear, *Formal Techniques in Real-Time and Fault Tolerant Systems, Proc. 6th International Symposium, FTRTFT 2000* Pune, India, September 20-22, 2000. Springer LNCS Vol. 1926. Copyright Springer-Verlag. Work supported by an Australian Research Council Large Grant, and by the Technion fund for advancement of research. Thanks to Kai Engelhardt for helpful discussions on the topic of this paper.

frequently need to refer not just to the current state, but also to past and future events in the system. Thus, this approach enriches the process algebra with features of modal logic, typically Hennessy-Milner logic or the more expressive μ-calculus. On the other hand, these works do not contain the assertions available in the state-based approaches.

Our ultimate goal is to develop a framework for distributed programs that has both the rich compositional structure and temporal expressiveness of the process calculus based approaches and the ability to represent the two types of annotation available in the predicate transformer based approaches. Moreover, we aim for a high degree of flexibility in the use of assertions: given the temporal setting, we would like to be able to reason about, and refine, any program fragment based on properties established by any other program fragment, both past and future. Our contribution in this paper is to develop such a framework for sequential programs.There exist prior proposals [UF96] combining the expressive power of temporal formalisms and the annotational expressiveness of the predicate transformer based approaches to refinement, but we argue that our framework allows for some desirable modes of reasoning not available in these proposals. In particular, we show by means of an example (in Section 8) that it is possible in our framework to move annotations within the program in a manner that is not supported by other formalisms.

2 Syntax

The setting we are considering is one in which there is a single agent that performs actions, which in turn modify the state. We assume as given an *environment*, consisting of a tuple (S, A, R, π), where A is a set of basic *actions*, S is a set of *states*, $R \subseteq S \times A \times S$ is a transition relation describing the effect of actions on the state, and $\pi : Prop \to \mathcal{P}(S)$ is an *interpretation*, mapping each propositional constant in some set *Prop* to a set of states. The environment will act as an implicit parameter in what follows.

Starting from a logical language $\mathcal{L}$ of formulas, which we shall define shortly, the set A of basic program actions, a set PV of *program variables* and a set CV of *constraint variables*, we define the class PT of *program templates* as follows.

$$P ::= \varepsilon \mid a \mid Z \mid [\varphi, \psi]^X \mid [\varphi]^X \mid \{\varphi\}_J \mid P;P \mid P+P \mid P^\omega$$

where ε is a distinguished symbol representing the empty program, $a \in A$ is a basic action, $Z \in PV$ is a program variable, φ and ψ are formulae of $\mathcal{L}$, $X \in CV$ is a constraint variable, and $J \subseteq CV$ is a set of constraint variables. A program template containing no program variables will be called simply a *program.* The program $[\varphi]^X$ is called a *coercion*, while $[\varphi, \psi]^X$ is called a *specification*. We call these constructs *constraints*. A program of the form $\{\varphi\}_J$ is called an *assertion*. In contrast to coercions, an assertion does not constrain the execution. It is a statement, akin to a comment in the program text, whose truth depends on the rest of the program. The subscript J is called the *justification* of the assertion, and it keeps track of what constraints in the current program this assertion depends on. Intuitively, a program is called *valid* if the assertions in the program are all guaranteed to be satisfied. We will present a detailed motivation for the constraint variables and justifications in Section 5.

The operator ; represents sequential execution of programs. The operators $+$, representing nondeterministic choice, and $^{\omega}$, representing finite or infinite repetition, are nondeterministic. Together with coercions, they can be used to define the standard deterministic branching and looping operators **if- then - else** and **while**. We will use the following definitions of these constructs.

– $\mathbf{if}^X\varphi$ **then** P **else** Q will stand for $([\varphi]^X;P) + ([\neg\varphi]^X;Q)$, and
– $\mathbf{while}^X\varphi$ **do** P will stand for $([\varphi]^X;P)^{\omega};[\neg\varphi]^X$.

To define constraints and assertions, we use a modal logical language $\mathcal{L}$. For the sake of brevity, we use a simple version of propositional linear-time temporal logic. This language can be extended considerably without losing the properties we shall discuss. Given a set *Prop* of primitive propositions, the formulas of our language $\mathcal{L}$ are inductively defined as follows:

Every proposition in *Prop* is a formula, and if φ_1 and φ_2 are formulas then so are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, and $\Box\varphi_1$. We define **true** as $\neg(P \wedge \neg P)$, **false** as $\neg$**true**, $\varphi_1 \vee \varphi_2$ as $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2$ as $\neg\varphi_1 \vee \varphi_2$, and $\Diamond\varphi$ as $\neg\Box\neg\varphi$, as usual.

3 Semantics

3.1 Semantics of Formulas

Semantically, formulas of the modal language describe what is true at time points of the executions of programs in the environment (S,A,R,π). A *run* over (S,A,R,π) is defined by a pair $r = (h,\alpha)$ where $h : \mathbf{N} \rightarrow S$ and $\alpha : \mathbf{N} \rightarrow A$ such that $(h(n),\alpha(n),h(n+1)) \in R$ for all $n \in \mathbf{N}$. The h component is a *state history*, describing the sequence of states the system goes through. The component α is the *action history*, describing the sequence of actions taken. Intuitively, we think of $h(k)$ as the state at time k, and $\alpha(k)$ as the action performed at time k. A *point* is a pair (r,k) where r is a run and $k \in \mathbf{N}$ represents a time.

Formulas are said to be true or false at a point (r,k). The fact that φ is true at (r,k) is denoted by $r,k \models \varphi$, and is defined by induction on the structure of formulas as follows (here and elsewhere, we leave the environment implicit):

1. $r,k \models p$ for $p \in Prop$ if $h(k) \in \pi(p)$, where $r = (h,\alpha)$.
2. $r,k \models \neg\varphi$ if $r,k \not\models \varphi$.
3. $r,k \models \varphi_1 \wedge \varphi_2$ if both $r,k \models \varphi_1$ and $r,k \models \varphi_2$.
4. $r,k \models \Box\varphi$ if $r,k' \models \varphi$ holds for all $k' \geq k$.

3.2 Semantics of Programs

We will develop the complete meaning of the program constructs in a number of steps. We begin in this section by presenting a semantics of programs that ignores the role of assertions and constraint variables: we will introduce the semantic notions related to these once we have motivated them from some requirements on our approach to refinement.[1]

[1] Viewed in isolation, the definitions of this section could be given a simpler presentation: we go into the complexities of execution trees to prepare us for the later definition of validity.

A *time interval* is a pair $[c,d]$ where c and d are elements of $\mathbf{N}^+ = \mathbf{N} \cup \{\infty\}$ with $c \leq d$. A *run interval* is a pair $r,[c,d]$ consisting of a run r and a time interval $[c,d]$. We will refer simply to "intervals"when the specific type of interval is clear from the context. We give semantics to programs in an operational style, by defining when an execution of a program P *occurs* over an interval $[c,d]$ in a run r, which we denote by $r,[c,d] \Vdash P$.

An *execution tree* for a program P is an ordered tree representing a possible execution of the program. Intuitively, such a tree describes one particular way in which the nondeterministic choices that may be made in running the program are resolved. The left to right ordering of nodes in a tree corresponds to precedence in time. Formally, an *execution tree* will be a finite or infinite tree T in which each node is adorned by a program template, subject to the following conditions on the nodes n of T:

1. If n is adorned by ε, by a program variable, a constraint, an assertion or a basic action, then n is a leaf.
2. If n is adorned by $P;Q$ then n has exactly two children, the leftmost adorned by P, the other adorned by Q.
3. If n is adorned by $P+Q$ then n has exactly one child, adorned by either P or Q.
4. If n is adorned by P^ω then n has exactly one child, adorned either by ε or by $P;(P^\omega)$.

An *execution tree for a program template P* is an execution tree whose root is adorned by P. We write $\mathcal{E}(P)$ for the set of execution trees of P.

Let T be an execution tree and $r = (h,\alpha)$ a run. We say that a mapping θ associating an interval with every node of T is an *embedding* of T in the interval $r,[c,d]$ if the following conditions are satisfied:

1. If n is the root of T then $\theta(n) = [c,d]$.
2. If n has a single child m then $\theta(n) = \theta(m)$.
3. If n has exactly two children m_1 and m_2 then there exists $e \leq f \leq g$ such that $\theta(m_1) = [e,f]$ and $\theta(m_2) = [f,g]$ and $\theta(n) = [e,g]$.
4. If n is adorned by the empty program ε and $\theta(n) = [e,f]$ then $e = f$.
5. If n is adorned by an assertion $\{\varphi\}_J$ and $\theta(n) = [e,f]$ then $e = f$.
6. If n is adorned by a basic action a and $\theta(n) = [e,f]$ then either $e = f = \infty$ or $e < \infty$, $f = e+1$, and $\alpha(e) = a$.
7. If n is adorned by a coercion $[\varphi]^X$ and $\theta(n) = [e,f]$ then $e = f$ and if $e < \infty$ then $r,e \models \varphi$.
8. If n is adorned by a specification $[\varphi,\psi]^X$ and $\theta(n) = [e,f]$ then if $e < \infty$ and $r,e \models \varphi$ then $f < \infty$ and $r,f \models \psi$.
9. If $\theta(n) = [e,f]$ then f is the least upper bound of the set of $g \in \mathbf{N}^+$ such that $\theta(m) = [g',g]$ for some *leaf* node m descended from n.

Notice that we made no special requirement about how θ should map a leaf adorned by a program variable $Z \in PV$. This is intentional, since in our view a program variable may be replaced by an arbitrary program, leading to an arbitrary mapping. In this sense, a program variable is treated just like a specification of the form $[\mathbf{false},\mathbf{true}]$.

To understand the need for condition (9) of the definition of embedding, let P be the program $([\mathbf{true}]^X)^\omega$ and consider the infinite tree $T \in \mathcal{E}(P)$ depicted in Figure 1.

(We omit the constraint variable X in this figure.) This tree depicts a computation in which a coercion is repeated infinitely often. Intuitively, as a coercion takes no time, this computation should take no time either. Suppose θ is an embedding of T into $r,[c,d]$. Thus, $\theta(\mathtt{root}(T)) = [c,d]$. It follows from conditions (2),(3) and (7) of the definition of embedding that $\theta(n) = [c,c]$ for all leaves n of T (each of which is adorned by $[\mathbf{true}]^X$). However, this still leaves open the possibility that $c < d$, which conflicts both with the intuition that the execution should take no time and with the intuition that the transition from time c to time d is effected by executing the sequence of statements represented at the leaves of the tree from left to right.

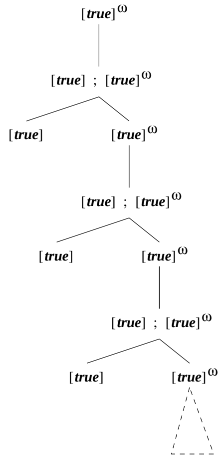

Fig. 1. A tree requiring a limit condition on embeddings

We write $r,[c,d] \Vdash_{T,\theta} P$ if $T \in \mathcal{E}(P)$ and θ is an embedding of T in the interval $r,[c,d]$. Intuitively, this means that T and θ describe an execution of P in this interval. We say that the program P *occurs* over $r,[c,d]$, denoted by $r,[c,d] \Vdash P$, if there exist T and θ such that $r,[c,d] \Vdash_{T,\theta} P$.

Define a program to be *concrete* if it does not contain assertions or specifications, it contains coercions only within **if** and **while** statements, and the formulas in these coercions contain no temporal operators. One can give a standard operational semantics to concrete programs, and establish its equivalence to the semantics presented above. We omit the details for reasons of space.

4 Refinement

In this section we consider some desiderata for the notion of program refinement for our framework. We propose some definitions in this section that are appropriate for programs not containing assertions. We will later need to adjust these definitions in order to accommodate assertions.

The notion of refinement we seek will be represented by means of a binary relation $\leq$ on programs, such that $P \leq Q$ when P is a refinement of Q. Intuitively, this means that the program P has less nondeterminism than Q, in the sense that every execution of P is an execution of Q. Recalling that our programs also function as specifications, an alternate way of phrasing this is that P carries more information, or is more constraining than Q. In order to support a reasonable calculus for refinement, we have the following desiderata for this refinement relation: it should be a reflexive, transitive relation satisfying

Monotonicity: If $P \leq Q$ and $C(Z)$ is a program template then $C(P) \leq C(Q)$.[2]
Reduction of Nondeterminism: $P \leq P+Q$ and $Q \leq P+Q$
Increase of Information: $[\varphi]^X \leq [\psi]^X$ if $\varphi \rightarrow \psi$ is valid.

We will say that a refinement relation is *adequate* if it satisfies the above conditions for programs not containing assertions.

Define a refinement relation $\sqsubseteq$ by having $P \sqsubseteq Q$ if for all intervals $r,[c,d]$, if $r,[c,d] \Vdash P$ then $r,[c,d] \Vdash Q$. We write $P \equiv Q$ when $P \sqsubseteq Q$ and $Q \sqsubseteq P$. It is not difficult to show that this definition satisfies all the properties above:

Theorem 1 *The binary relation $\sqsubseteq$ on program templates is adequate.*

Example 1. Consider a program that is to access a shared resource such as a printer, which is guarded by a locking mechanism. Only an application that has a lock on the resource can use it. Moreover, we wish to ensure that the program does not hold on to the lock indefinitely. We model this by a proposition `have_lock` which is defined to be true when the application has obtained a lock on the resource, and three (terminating) actions: get_lock, release_lock, and use_resource, corresponding to the action of fetching the lock, releasing the lock, and using the resource. Given an appropriate definition of the environment (omitted), the first two actions have the property that

$$\mathsf{get\_lock} \sqsubseteq [\textbf{true}, \texttt{have\_lock}]^U \quad \text{and} \quad \mathsf{release\_lock} \sqsubseteq [\textbf{true}, \neg\texttt{have\_lock}]^U$$

for any constraint variable U. Moreover,

$$\mathsf{use\_resource} \sqsubseteq [\texttt{have\_lock}, \texttt{have\_lock}]^U$$

so that if started in a state where the application has the lock, the use of the resource will terminate in finite time, and does not relinquish the lock.

The restrictions on the use of the resource by the application can be guaranteed by requiring that the resource can only be used via a *legal access* procedure, defined by:

$$Access(X,Y) = [\textbf{true}, \texttt{have\_lock}]^X ; \mathsf{use\_resource} ; [\textbf{true}, \Diamond\neg\texttt{have\_lock}]^Y$$

Note that this allows that after using the resource the program need not release the lock immediately. However, it must ensure that the lock is eventually released. In a followup example in Section 8 we will demonstrate how this is possible. We now turn to implementing $Access(X,Y)$. From the fact that $\neg\texttt{have\_lock} \rightarrow \Diamond\neg\texttt{have\_lock}$ is a valid formula and the assumption that $\mathsf{release\_lock} \sqsubseteq [\textbf{true}, \neg\texttt{have\_lock}]^U$ we can show that $\mathsf{release\_lock} \sqsubseteq [\textbf{true}, \Diamond\neg\texttt{have\_lock}]^U$. Monotonicity of $\sqsubseteq$ now yields

$$(\mathsf{get\_lock} ; \mathsf{use\_resource} ; \mathsf{release\_lock}) \sqsubseteq Access(X,Y),$$

giving us one straightforward way of implementing *Access*.

[2] We use the convention that if $C(Z)$ is a program template containing the program variable Z then $C(P)$ is the program template obtained by replacing every occurrence of Z in $C(Z)$ by P.

The following example illustrates a sense in which our sequential framework is already able to capture some aspects of a distributed setting.

Example 2. The previous example treats the action of obtaining the lock as taking a single time step. This makes sense if time steps represent state transitions for the program we are developing, but if the program operates concurrently with other programs that may hold a lock on the resource, then we may wish to reason about the period during which the program waits for a lock held by another process to be released. This may be done by introducing another proposition $\mathtt{have\_lock}_e$ to represent that the environment of the program holds the lock, and an atomic action `lock` such that $\mathtt{lock} \sqsubseteq [\neg\mathtt{have\_lock}_e, \mathtt{have\_lock}]^X$, as well as an atomic action `skip` that does not change the value of `have_lock` but may switch the value of $\mathtt{have\_lock}_e$. We may then implement the specification $[\mathbf{true}, \mathtt{have\_lock}]^U$ by the refinement

$$[\Box(\mathtt{have\_lock}_e \rightarrow \Diamond\neg\mathtt{have\_lock}_e)]^X; (\mathbf{while}^Y\, \mathtt{have\_lock}_e\ \mathbf{do}\ \mathtt{skip}); \mathtt{lock} \sqsubseteq [\mathbf{true}, \mathtt{have\_lock}]^U.$$

Note that the coercion in this program states that the environment of the program is required to eventually release any lock it may hold — this ensures eventual termination of the while loop and success of the `lock` action. (We could eliminate the need for such coercions by generalizing environments to include liveness conditions, but we will not pursue this here.)

5 Reasoning with Assertions

In deriving programs by refinement, it is frequently the case that one part of the program being developed ensures conditions that can be used to optimize other parts of the program. To take advantage of such opportunities for optimization, it is useful to have in a refinement calculus a facility for making assertions that are derivable from program fragments, and rules that allow such assertions to be exploited in refinement steps. The assertions $\{\phi\}_C$ in our framework are intended to play such a role. A further desideratum for our framework is that it should support reasoning about the temporal properties guaranteed by a program fragment based on the properties of program fragments that will run in the future. (We present an example of such reasoning later.) We now explain how this desideratum motivates our use of constraint variables and justifications. We suppose for the sake of the argument that constraints and assertions do not have these annotations.

Consider the program $[\mathbf{true}, \phi]$. When this program terminates (as it must) ϕ will be true. Thus, for a refinement relation $\leq$ that supports the introduction of assertions into the program text, we would like the rule $[\mathbf{true}, \phi]; \{\phi\} \leq [\mathbf{true}, \phi]$ to be sound. Intuitively, $\{\phi\}$ asserts that ϕ is guaranteed to hold at this location. Moreover, we would like to be able to exploit assertions to perform refinements. When $[\mathbf{true}, \phi]; \{\phi\}$ occurs within the context of a larger program, the reason for the assertion $\{\phi\}$ may be some following part of the program that guarantees that ϕ will hold at this location. (For example, ϕ might assert that a message will eventually be delivered, and the following code may send this message on a reliable channel.) In this situation, we would like to be able to use this fact to simplify the specification $[\mathbf{true}, \phi]$ to ε. Intuitively, this program fragment need do nothing, since its intended effect is taken care of by some other part of the program. This suggests that we want the rule $\varepsilon \leq [\mathbf{true}, \phi]; \{\phi\}$ to be sound.

Of course, we cannot have both these rules, since we would obtain by transitivity that $\varepsilon \le [\mathbf{true}, \phi]$, which clearly cannot be sound. What has gone wrong is that we have attempted to refine a program fragment based on an assertion derived from that very program fragment. This is circular reasoning! To block such circularities, we keep track of the constraints that the truth of an assertion depends on, and allow refinement steps of this type only when they do not involve circular reasoning. We do this by labeling constraints by constraint variables, providing assertions with justifications, and carefully tracking the way that assertions depend on constraints by appropriately adjusting the justifications when refinement steps are performed. Intuitively, an assertion $\{\varphi\}_J$ in a program amounts to a claim that φ holds, with the proof depending on the fact that the requirements implied by the constraints associated with all $X \in J$ are satisfied. For example, if $Y \in J$ and $[\psi_1, \psi_2]^Y$ occurs in the program, then the proof of φ may depend on the segment of code that the designer ultimately substitutes for $[\psi_1, \psi_2]^Y$ having the property that if started in a state satisfying ψ_1, it will terminate in a state satisfying ψ_2.

Using this idea, the two refinements above become $[\mathbf{true}, \phi]^X; \{\phi\}_{\{X\}} \le [\mathbf{true}, \phi]^X$, and $\varepsilon \le [\mathbf{true}, \phi]^X; \{\phi\}_J$ *provided* $X \notin J$. The latter still allows us to perform the optimization when the assertion is justified by some other part of the program, while blocking the undesirable refinement above.

The decision to track justifications of assertions and allow rules such as

$$\text{“}\varepsilon \le [\mathbf{true}, \phi]^X; \{\phi\}_J \quad \text{provided} \quad X \notin J\text{”}$$

leads to some further complexities. Note that this rule eliminates the constraint $[\mathbf{true}, \phi]^X$. The larger program within which this refinement occurs may contain assertions that depend on this constraint. These assertions are still valid, but the reason for the validity is now that ε runs in place of $[\mathbf{true}, \phi]^X$, and that this constitutes a correct implementation of $[\mathbf{true}, \phi]^X$ because of the condition ϕ guaranteed by the constraints in J. That is, J becomes part of the explanation for assertions that previously depended on X. This means that in applying this rule, we need to transform the justifications for assertions elsewhere in the program.

To formalize these transformations, define a *justification transformation* to be a mapping $\eta : \mathcal{P}(CV) \to \mathcal{P}(CV)$ that is *increasing*, i.e., satisfies $J \subseteq \eta(J)$ for all $J \subseteq CV$. The result of applying a justification transformation η to a program P (or, respectively, program template $C(Z)$), is the program $P\eta$ (respectively, the program template $C\eta(Z)$) obtained by replacing every assertion $\{\varphi\}_J$ in P (respectively, in C), by the corresponding assertion $\{\varphi\}_{\eta(J)}$ in which its justification set has been transformed by η. We write $C\eta(P)$ for the program obtained by substituting P for Z in $C\eta(Z)$.

The assumption that justification transformations are increasing arises from the fact that we allow more than one constraint to be associated with the same constraint variable. (Programs in which this is the case arise naturally from desirable program transformations, such as refining P^ω by $P;P^\omega$. This transformation would create copies of any constraints in P, resulting in multiple occurrences of constraint variables.) Consider a program such as

$$\{twice(\varphi)\}_{\{X\}} \ ; \ [\varphi]^Y \ ; \ \{\varphi\}_{\{Y\}} \ ; \ [\varphi]^X \ ; \ a \ ; \ [\varphi]^X$$

where a is any atomic action and $twice(\varphi)$ expresses that there are at least two distinct time points in the future at which φ holds. This program is valid. If we were to apply the coercion elimination rule discussed above, and apply a justification transformation η mapping $\{X\}$ to $\{Y\}$, we would obtain the program

$$P \;=\; \{twice(\varphi)\}_{\{Y\}}\ ;\ [\varphi]^Y\ ;\ \varepsilon\ ;\ a\ ;\ [\varphi]^X$$

This program is not valid, because it states $\{Y\}$ as the justification set for the truth of the assertion $twice(\varphi)$. The program $P^{\{Y\}}$, which is obtained from P by replacing the coercion $[\varphi]^X$ by ε has executions in which φ holds only once and $twice(\varphi)$ is falsified. A sound refinement step would have stated that the assertion $twice(\varphi)$ depends on both X and Y. Hence, the appropriate justification transformation here is to map $\{X\}$ to $\{X,Y\}$; the elimination of a coercion $[\varphi]^X$ introduced a dependency on Y, but additional dependency on X still needs to be accounted for. More generally, when transforming justifications in refining a constraint with label X, we need to preserve instances of X, since occurrences of X in assertions may be due to X-constraints other than the one being refined.

It is convenient to introduce some notation for particular justification transformations that will occur frequently in what follows. The identity justification transformation, in which $\eta(J) = J$ for every $J \subseteq CV$, is denoted by ι. We will also represent justification transformations using expressions of the form $[X_1 \hookrightarrow S_1, \ldots, X_n \hookrightarrow S_n]$, where the $X_i \in CV$ are constraint variables and the $S_i \subseteq CV$ are sets of constraint variables. (In the simple case of $n = 1$, we write $X \hookrightarrow S$ instead of $[X \hookrightarrow S]$, and if S is a singleton $\{Y\}$ we write $X \hookrightarrow Y$.) Such an expression denotes the justification transformation η defined by $\eta(J) = J \cup \bigcup_{X_i \in J, i=1\ldots n} S_i$ for all $J \subseteq CV$. It is also useful to talk about the *composition* of justification transformations. We define the composition $\eta \cdot \eta'$ by $\eta \cdot \eta'(J) \;=\; \eta(\eta'(J))$ for all $J \subseteq CV$.

6 Validity and Valid Refinement

We now provide a formal semantics for the ideas motivated in the previous section.

6.1 Validity

The semantics of assertions $\{\varphi\}_J$ can be made precise as follows. First, to help capture the dependency of the assertion on J, we define a program transformation that modifies a program to one in which the program structure is maintained, but only the constraints labelled by a variable in a given justification set are enforced. Let J be a justification set and P a program. We define the program P^J by a recursion on the structure of P. In the cases where P is the null program ε, an atomic action a, a program variable Z, or an assertion $\{\psi\}_{J'}$, we define $P^J = P$. The other base cases concern program constraints. In the case of coercions, we define $([\varphi]^X)^J$ to be $[\varphi]^X$ if $X \in J$ and ε otherwise. That is, if the constraint variable labeling a coercion is in the set J, then the coercion is preserved, otherwise it is eliminated. Note that ε and the coercion it replaces are programs requiring no time to execute, so the temporal behaviour of the program is not modified by this substitution. In the case of specifications, we define $([\varphi,\psi]^X)^J$ to be $[\varphi,\psi]^X$ if $X \in J$ and $[\mathbf{false},\mathbf{true}]^X$ otherwise. Again, if the constraint variable X labeling the specification is in the set J, then the specification is preserved. If $X \notin J$, then the specification is replaced by $[\mathbf{false},\mathbf{true}]^X$, which is the specification that does not impose any requirements. Intuitively, this specification may run for any finite or infinite amount of time, performing an arbitrary sequence of actions. More formally, it can be seen from the definition of occurrence that $r,[c,d] \Vdash [\mathbf{false},\mathbf{true}]^X$ holds for *every* interval $r,[c,d]$.

The recursive cases of the definition of P^J are given by: (i) $(P;Q)^J = P^J;Q^J$, (ii) $(P+Q)^J = P^J + Q^J$ and (iii) $(P^\omega)^J = (P^J)^\omega$. Intuitively, these cases preserve the structure of the program and filter the transformation down to the base cases.

Notice that in moving from P to P^J we are *coarsening* the program:

Lemma 1. *For every program P and justification set J,* $\quad P \sqsubseteq P^J$.

We define a program P to be *valid* if the following condition holds: for every assertion $\{\varphi\}_J$ appearing in P, for all r, $[c,d]$, T, θ, n and e, if (i) $r,[c,d] \Vdash_{T,\theta} P^J$, (ii) n is a node of T adorned by $\{\varphi\}_J$ and (iii) $\theta(n) = [e,e]$ where $e < \infty$, then $r,e \models \varphi$.

Intuitively, this condition holds when the assertion φ holds at each point of an execution of the program P^J that corresponds to a location in the text of P^J at which the assertion $\{\varphi\}_J$ occurs. Note that we check assertions only at finite times, and do not require for validity that all assertions of the program be reached during the execution. In this respect, our notion of validity resembles *partial correctness* of program assertions [Hoa67].

Assuming the standard semantics for the assignment statement and the obvious interpretations of the propositions, the following is an example of a valid program:

$$P_1 \quad = \quad [x=0]^X \; ; \; \{\Diamond(y>1)\}_{\{X\}} \; ; \; y \leftarrow x+1 \; ; \; y \leftarrow y+1 \; ; \; \{y=2\}_{\{X\}}$$

The justification of the assertions in this program is the singleton set $\{X\}$. The truth of the assertions in this program depends on more than just the coercion $[x=0]^X$, but the other parts it depends on are concrete program segments, which cannot be modified at a later stage of a refinement process.

Program P_2 is another valid program, illustrating the sense in which validity represents a notion of partial correctness:

$$P_2 \quad = \quad [x=0]^X \; ; \; (\mathbf{while}^Y \; \mathbf{true} \;\; \mathbf{do} \; x \leftarrow x+1) \; ; \{\mathbf{false}\}_{\{Y\}}$$

To see that this program is valid, notice that $\{\mathbf{false}\}_{\{Y\}}$ is the only assertion in P_2. In all execution trees of $P_2^{\{Y\}}$ that can be embedded in an interval $r,[c,d]$, the node adorned by this assertion is mapped to infinity, because it is preceded by infinitely many time steps in which x is incremented. The assertion is thus vacuously satisfied at all finite points that its nodes are mapped to, and it follows that P_2 is valid. Notice that this example also shows that the general problem of deciding whether a program is valid is at least as hard as proving nontermination of programs in our language.

The slight variant $\quad [x=0]^X \; ; \; (\mathbf{while}^Y \; \mathbf{true} \;\; \mathbf{do} \; x \leftarrow x+1) \; ; \{\mathbf{false}\}_{\{X\}}$ is not a valid program, however.

6.2 Valid Refinement

We are now ready to define a validity-preserving notion of refinement. As noted above, we need to apply a justification transformation η to the surrounding context in some cases. We will write $P \preceq_\eta Q$ to state that a program template P validly refines a program template Q, subject to a justification transformation η to be applied to the syntactic context in which the refinement is to be applied. We define $P \preceq_\eta Q$ to hold if for all program templates $C(Z)$, if $C(Q)$ is valid then (i) $C\eta(P) \sqsubseteq C(Q)$, and (ii) $C\eta(P)$ is valid. Notice that the transformation η affects only the form of assertions, which are effectively ignored by the refinement relation $\sqsubseteq$. As a result, $P \equiv P\eta$ and $C(P) \equiv C\eta(P)$ hold for every C, P, and η. Thus, condition (i) in the definition of valid refinement can equivalently be stated as (i') $C(P) \sqsubseteq C(Q)$. Moreover, since $\sqsubseteq$ is monotone, we can often obtain (i') from $P \sqsubseteq Q$. As we will see, however, there are important cases in which $P \preceq_\eta Q$ in cases where $P \not\sqsubseteq Q$.

Clearly, for every program template $C(Z)$, if $P \preceq_\eta Q$ and $C(Q)$ is valid then $C(P) \sqsubseteq C(Q)$. Thus, $P \preceq_\eta Q$ states in part that it is possible to obtain a refinement of any valid program by locally substituting P for Q. In addition to this, the definition requires that this substitution be validity preserving, provided we transform the justifications in the context by η. The following result shows that this relation has the transitivity and monotonicity properties, provided that we make appropriate allowance for the justification transformations.

Theorem 2 *The relations $\preceq_\eta$ satisfy the following:*

1. *If $P \preceq_\eta Q$ and $Q \preceq_{\eta'} R$ then $P \preceq_{\eta \cdot \eta'} R$.*
2. *If $P \preceq_\eta Q$ then $C\eta(P) \preceq_\eta C(Q)$ for all program templates $C(X)$.*

An easy corollary of Theorem 2 is that $\preceq_\iota$ is an adequate refinement relation. We write $P \preceq Q$ for $P \preceq_\iota Q$, and write $P \asymp Q$ when both $P \preceq_\iota Q$ and $Q \preceq_\iota P$.

The following example illustrates how assertions combine with the notion of valid refinement to enable refinement steps that exploit properties of the context in which the refinement takes place.

Example 3. In example 2, we refined a specification to a concrete program together with a coercion that states a property that the environnment must have for this concrete program to implement the specification. We may restate this example using valid refinement as follows:

$$(\mathbf{while}^Y\, \mathtt{have\_lock}_e\ \mathbf{do}\ \mathtt{skip}); \mathtt{lock} \preceq \{\Box(\mathtt{have\_lock}_e \to \Diamond\neg\mathtt{have\_lock}_e)\}_J; [\mathbf{true}, \mathtt{have\_lock}]^X$$

provided $X \notin J$. Intuitively, this states that the refinement is valid provided it is done in a context in which one can prove that the environment is guaranteed to eventually release the lock.

7 Rules for Valid Refinement

We now present a number of refinement rules sound with respect to the above semantics. (This is not a complete list of rules for our semantics.) First, we have a rule which enables one of the most basic steps used in top-down design of sequential programs (cf. [Mor90]):

Sequential composition: $[\varphi,\varphi_1]^X;\ [\varphi_1,\psi]^Y \ \preceq_{U\hookrightarrow\{X,Y\}}\ [\varphi,\psi]^U$

Many of the properties of $\sqsubseteq$ translate into valid refinements with the identity transformation:

Identity: $\varepsilon;P \asymp P$ and $P;\varepsilon \asymp P$

ω-rules: $P^\omega \asymp (\varepsilon+P;P^\omega)$ and $P^\omega;P^\omega \asymp P^\omega$

Associativity: $(P;Q);R \asymp P;(Q;R)$ and $(P+Q)+R \asymp P+(Q+R)$

Commutativity and Idempotence of $+$: $P+Q \asymp Q+P$ and $P+P \asymp P$

Distribution: $P;(Q+R) \asymp (P;Q)+(P;R)$ and $(Q+R);P \asymp (Q;P)+(R;P)$

We say that a program is *vanishing* if it is the empty program ε, a coercion $[\varphi]^X$ or an assertion $\{\varphi\}_J$. These programs are called vanishing because they occur over point intervals and take no time to execute. The rules for vanishing programs do not hold in the formalism of Morgan [Mor90], for example.

Commutativity and Idempotence of vanishing programs: for all vanishing P,Q, we have $P;Q \asymp Q;P$ and $P;P^\omega \asymp P$

The examples just given demonstrate standard refinements that still hold as valid refinements. Not every standard refinement translates into a valid refinement. For example, we clearly have $\{\textbf{false}\} \sqsubseteq \varepsilon$ but not $\{\textbf{false}\} \preceq_\iota \varepsilon$, since for the template $C(Z)=Z$ we have that $C(\varepsilon)$ is a valid program, while $C(\{\textbf{false}\})$ is not. But even for programs that do not contain assertions, one needs to be careful. For example, we have $[\varphi]^X \sqsubseteq [\varphi]^Y$. This does correspond to a valid refinement, but one in which anything that depends on Y should be made to depend on both X and Y. Thus, we have the valid refinement rule

Renaming Constraint: $[\varphi]^X \preceq_{Y\hookrightarrow X} [\varphi]^Y$ and $[\varphi,\psi]^X \preceq_{Y\hookrightarrow X} [\varphi,\psi]^Y$

A major reason for using valid refinement is for proving that $P \preceq_\eta Q$ in cases when $P \sqsubseteq Q$ *does not* hold. Intuitively, this will be the case for refinements that exploit properties of the context in which the refinement step is taking place. Examples of this are:

Coercion Elimination: $\varepsilon \ \preceq_{X\hookrightarrow J}\ \{\varphi\}_J;[\varphi]^X$ provided $X \notin J$

Specification Elimination: $\varepsilon \ \preceq_{X\hookrightarrow J}\ [\varphi,\psi]^X\ ;\ \{\varphi\rightarrow\psi\}_J$ and $\varepsilon \preceq_{X\hookrightarrow J}\ \{\varphi\rightarrow\psi\}_J\ ;\ [\varphi,\psi]^X$, provided $X \notin J$

Of special interest are rules that allow us to move temporal assertions around the program text. One example is

Advance Box: $P;\{\Box\varphi\}_J \preceq \{\Box\varphi\}_J;P$

which moves a temporal assertion forward in time. It is also possible to move temporal assertions backwards.

Regress Diamond: If $P \preceq_{X\hookrightarrow J_1} [\textbf{true},\textbf{true}]^X$ then $\{\Diamond\varphi\}_{J_1\cup J_2};P \preceq P;\{\Diamond\varphi\}_{J_2}$

We remark that the statement $P \preceq_{X\hookrightarrow J_1} [\textbf{true},\textbf{true}]^X$ ensures that P is a halting program, and so it provides one approach to expressing termination.

A few additional properties of $\preceq$ that we shall use in the following example are

Strengthen Spec: If $\varphi_1 \to \varphi$ and $\psi \to \psi_1$ are valid formulae of the logic of $\mathcal{L}$ then $[\varphi,\psi]^X \preceq_{Y \hookrightarrow X} [\varphi_1,\psi_1]^Y$

Specification Consequence: $[\varphi,\psi]^X \,;\, \{\psi\}_{J\cup\{X\}} \preceq \{\varphi\}_J \,;\, [\varphi,\psi]^X$

Valid Assertion: If φ is a valid formula of the logic of $\mathcal{L}$ then $\{\varphi\}_\emptyset \asymp \varepsilon$.

As mentioned above, some of our rules are fairly standard, and appear in other refinement calculi. In the example presented in the next section we make effective use of two rules which do not hold in typical systems: The Regress Diamond and Specification Elimination rules.

8 Optimization using refinement

We now consider an example that illustrates some of the rules mentioned above and shows that our framework differs from others in useful ways. In the setting of Example 1 from Section 4, consider a case in which the resource needs to be used twice in a row. Thus, our goal is to implement the program *Access_twice* defined by

$$Access\_twice \quad = \quad Access(X1,Y1)\,;\,Access(X2,Y2).$$

Recall from Example 1 that

$$Access(X2,Y2) \quad = \quad [\mathbf{true},\texttt{have\_lock}]^{X2}\,;\,\mathsf{use\_resource}\,;\,[\mathbf{true},\Diamond\neg\texttt{have\_lock}]^{Y2}.$$

In Example 1 we argued that

$$(\mathsf{get\_lock}\,;\,\mathsf{use\_resource}\,;\,\mathsf{release\_lock}) \sqsubseteq Access(X2,Y2),$$

A similar valid refinement can be established:

$$(\mathsf{get\_lock}\,;\,\mathsf{use\_resource}\,;\,\mathsf{release\_lock}) \preceq Access(X2,Y2).$$

Indeed, one way to implement *Access_twice* would be to simply repeat this program twice in sequence. It is possible to do better, however. We now demonstrate how, with the aid of valid refinement, our framework can be used to obtain a more efficient implementation.

Let us first consider the final step of $Access(X2,Y2)$, i.e., the program $[\mathbf{true},\Diamond\neg\texttt{have\_lock}]^{Y2}$. Note that by using Identity and Valid Assertion (since **true** is a valid formula) we can obtain

$$\{\mathbf{true}\}_\emptyset;[\mathbf{true},\Diamond\neg\texttt{have\_lock}]^{Y2} \preceq \varepsilon;[\mathbf{true},\Diamond\neg\texttt{have\_lock}]^{Y2} \preceq [\mathbf{true},\Diamond\neg\texttt{have\_lock}]^{Y2},$$

and the Specification Consequence rule yields

$$[\mathbf{true},\Diamond\neg\texttt{have\_lock}]^{Y2};\{\Diamond\neg\texttt{have\_lock}\}_{\{Y2\}} \preceq [\mathbf{true},\Diamond\neg\texttt{have\_lock}]^{Y2}.$$

By Monotonicity we thus have that

$$Access(X2,Y2);\{\Diamond\neg\texttt{have\_lock}\}_{\{Y2\}} \preceq Access(X2,Y2).$$

One of the strong points of our framework is that we can now use our rules to move the temporal assertion to the beginning of $Access(X2,Y2)$! The idea is to apply the Regress Diamond rule three times and thereby move the assertion back over each of the components of $Access(X2,Y2)$.

Notice that by Strengthen Spec we have[3]

$$[\mathbf{true}, \Diamond\neg\mathtt{have\_lock}]^{Y2} \;\preceq\; [\mathbf{true},\mathbf{true}]^{Y2}.$$

Hence, by Regress Diamond we have that

$$\{\Diamond\neg\mathtt{have\_lock}\}_{\{Y2\}};[\mathbf{true}, \Diamond\neg\mathtt{have\_lock}]^{Y2} \;\preceq\; [\mathbf{true}, \Diamond\neg\mathtt{have\_lock}]^{Y2};\{\Diamond\neg\mathtt{have\_lock}\}_{\{Y2\}},$$

so we can move the temporal assertion back over the last component of $Access(X2,Y2)$. Since use_resource is a basic (terminating) action, we have use_resource $\preceq$ $[\mathbf{true},\mathbf{true}]^{Y2}$. Thus, again by Regress Diamond we can derive

$$\{\Diamond\neg\mathtt{have\_lock}\}_{\{Y2\}};\mathsf{use\_resource} \;\preceq\; \mathsf{use\_resource};\{\Diamond\neg\mathtt{have\_lock}\}_{\{Y2\}},$$

Finally, we again apply Strengthen Spec to obtain that

$$[\mathbf{true},\mathtt{have\_lock}]^{X2} \;\preceq\; [\mathbf{true},\mathbf{true}]^{X2}$$

and hence

$$\{\Diamond\neg\mathtt{have\_lock}\}_{\{X2,Y2\}};[\mathbf{true},\mathtt{have\_lock}]^{X2} \;\preceq\; [\mathbf{true},\mathtt{have\_lock}]^{X2};\{\Diamond\neg\mathtt{have\_lock}\}_{\{Y2\}}.$$

The end result of moving the temporal assertion back over these three clauses yields:

$$\{\Diamond\neg\mathtt{have\_lock}\}_{\{X2,Y2\}};Access(X2,Y2) \;\preceq\; Access(X2,Y2).$$

By monotonicity, it follows that

$$Access(X1,Y1);\{\Diamond\neg\mathtt{have\_lock}\}_{\{X2,Y2\}};Access(X2,Y2) \;\preceq\; Access\_twice.$$

Let us now focus on the interaction of the assertion in this program with the final statement in $Access(X1,Y1)$. Since $Y1 \notin \{X2,Y2\}$, we have by Specification Elimination that

$$\varepsilon \;\preceq_{Y1\hookrightarrow\{X2,Y2\}}\; [\mathbf{true}, \Diamond\neg\mathtt{have\_lock}]^{Y1};\{\Diamond\neg\mathtt{have\_lock}\}_{\{X2,Y2\}}$$

so that we can eliminate the third clause in $Access(X1,Y1)$ and derive

$$[\mathbf{true},\mathtt{have\_lock}]^{X1} \;;\; \mathsf{use\_resource} \;;\; Access(X2,Y2) \;\preceq_{Y1\hookrightarrow\{X2,Y2\}}\; Access\_twice.$$

One further improvement is now possible. As we did for the final step of $Access(X2,Y2)$, we can apply the Identity, Valid Assertion and Specification Consequence rules to $[\mathbf{true},\mathtt{have\_lock}]^{X1}$—the first step of $Access(X1,Y1)$, and obtain:

$$[\mathbf{true},\mathtt{have\_lock}]^{X1};\{\mathtt{have\_lock}\}_{\{X1\}} \preceq [\mathbf{true},\mathtt{have\_lock}]^{X1}.$$

[3] Strictly speaking, applying the rule we should write $\preceq_{Y2\hookrightarrow Y2}$ in the valid refinement, but it is easy to check that the justification transformation $Y2 \hookrightarrow Y2$ is the identity transformation.

With an additional application of Specification Consequence, using the property

$$\mathsf{use\_resource} \;\preceq\; [\mathtt{have\_lock}, \mathtt{have\_lock}]^{X1}$$

which was one of the given assumptions, we can derive

$$[\mathbf{true}, \mathtt{have\_lock}]^{X1}; \mathsf{use\_resource}; \{\mathtt{have\_lock}\}_{\{X1\}} \preceq [\mathbf{true}, \mathtt{have\_lock}]^{X1}; \mathsf{use\_resource}.$$

We can now apply the assertion $\{\mathtt{have\_lock}\}_{\{X1\}}$ to the first statement in $Access(X2,Y2)$, using Specification Elimination to obtain

$$\varepsilon \;\preceq_{X2 \hookrightarrow X1}\; \{\mathtt{have\_lock}\}_{\{X1\}}; [\mathbf{true}, \mathtt{have\_lock}]^{X2},$$

and we can use this to derive that

$$[\mathbf{true}, \mathtt{have\_lock}]^{X1}; \mathsf{use\_resource}; \mathsf{use\_resource}\ ;\ [\mathbf{true}, \Diamond\neg\mathtt{have\_lock}]^{Y2} \;\preceq_\eta\; \mathit{Access\_twice}$$

where $\eta = [X2 \hookrightarrow X1] \cdot [Y1 \hookrightarrow \{X2,Y2\}] = [X2 \hookrightarrow X1,\ Y1 \hookrightarrow \{X1,X2,Y2\}]$. The first and last steps of this program may be refined to get_lock and release_lock, respectively, as before, yielding a concrete implementation of *Access_twice*:

$$\mathsf{get\_lock}\ ;\ \mathsf{use\_resource}\ ;\ \mathsf{use\_resource}\ ;\ \mathsf{release\_lock} \;\preceq_\eta\; \mathit{Access\_twice}.$$

Note that we have achieved an interesting optimization by means of this reasoning: instead of releasing the lock after the first use and then re-acquiring it immediately, we simply hold onto the lock and use the resource twice. This optimization made essential use of valid refinement. Moreover, moving *temporal* assertions (both forward and backward) in the program text played an important role.

9 Conclusion

As discussed in the introduction, there exist many approaches to program refinement. Perhaps the work most closely related to ours is the work on real-time refinement originated by Utting and Fidge [UF96,UF97,HU97,Hay98]. This work shares our objective of developing a Back/Morgan-style framework with temporal features. In particular, Utting and Fidge define a real time specification construct $\star[\varphi,\psi]$, similar to our $[\varphi,\psi]$ in that φ and ψ may be formulae expressing properties not just of the current state, but also of the run. They give semantics to this construct by translation to an instance of Morgan's refinement calculus in which states are taken to correspond to what we have called "points" in this paper, i.e. a run together with a "current time" variable. The construct $\star[\varphi,\psi]$ is represented by a predicate transformer over such states that "changes only the current time." The resulting calculus is like ours in many ways. A crucial difference, however, is that it does not support refinements like $\{\Diamond\varphi\}; \star[\mathbf{true},\varphi] \;\preceq\; \star[\mathbf{true},\varphi]$ which are essential to the example in Section 8. Their framework does not make use of explicit justifications, but, intuitively, it avoids circular temporal refinements like that discussed in Section 5 by allowing assertions to be "justified" only by constraints that have "executed" at an earlier time. Thus, in the refinement above, while φ can be asserted after the specification $\star[\mathbf{true},\varphi]$, we cannot assert $\Diamond\varphi$ before it.

We believe that examples similar to that of Section 8 will be quite significant in practice. For example, it is common to optimize protocols in distributed systems by noting that in certain circumstances, one can omit sending a message because its intended effect is already guaranteed to occur (perhaps because of actions and events due to occur in the future). This is particularly useful when developing protocols that optimize use of bandwidth by minimizing the number of messages sent. Of course, our framework needs generalization to be applied to such protocols — this is a topic of current work; we have discussed considerations applying to the generalization elsewhere [MM98]. The example in Section 8 already suggests that mechanical assistance for the bookkeeping involved in maintaining and updating justification sets and justification transformations. In addition, we believe that results derived using temporal logic theorem provers and related tools can be incorporated in derivations in our framework by providing input into rules such as Strengthen Spec and Valid Assertion. There is clearly a wide range of topics and issues to explore in developing this topic.

The definition of valid refinement we have introduced here demonstrates that it is possible to develop a refinement calculus involving temporal assertions that has an easily understood semantics and supports some quite intuitive reasoning. One point is worth making however: because it involves a quantification over the syntactic contexts $C(\cdot)$, our definition of valid refinement is likely to be sensitive to the syntax of programs. It is therefore worthwhile to consider more semantic notions of valid refinement. In a followup paper we will define a semantic counterpart of valid refinement, and prove the soundness of the rules used in this paper, as well as a host of other rules, with respect to this semantic notion.

We are also in the process of developing the framework to deal with several extensions of the language studied here. One extension introduces a way of placing labels within a program and enriches the assertion language to allow it to express properties of the program locations corresponding to these labels. This allows some interesting properties such as termination to be expressed directly in the programming language (rather than indirectly, through a statement about refinement, as we have done here in the Regress Diamond rule). Use of labels also provides a tighter coupling between program structure and temporal assertions that enables some quite useful refinement rules. Finally, we are also in the process of extending the framework developed here to include quantification over "local predicates" [EvdMM98,EvdMM00]: this provides a framework generalizing knowledge-based [FHMV95] and knowledge-oriented programs [MK93].

References

[BvW98] R. J. Back and von Wright. *Refinement Calculus: A systematic approach.* Graduate Texts in Computer Science. Springer Verlag, 1998.

[EvdMM98] K. Engelhardt, R. van der Meyden, and Y. Moses. Knowledge and the logic of local propositions. In I. Gilboa, editor, *Proc. Conf on Theoretical Aspects of Reasoning about Knowledge*, pages 29–41. Morgan Kauffman, July 1998.

[EvdMM00] K. Engelhardt, R. van der Meyden, and Y. Moses. A refinement framework supporting reasoning about knowledge and time. In J. Tiuryn, editor, *Proc. of Founda-*

tions of Software Science and Computation Structures, Lecture Notes in Computer Science, pages 114–129. Springer Verlag, March 2000.

[FHMV95] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass., 1995.

[GS86] S. Graf and J. Sifakis. A logic for the description of non-deterministic programs and their properties. *Information and Control*, 68(1–3):254–270, January/February/March 1986.

[Hay98] I. Hayes. Separating timing and calculation in real-time refinement. In J. Grundy et al, editor, *International Refinement Workshop & Formal Methods Pacific, Proc. IRW/FMP'98*, Series in Discrete Mathematics and Theoretical Computer Science, 1998.

[HL95] K. Havelund and K. Larsen. A refinement logic for the fork calculus. In S. T. Vuong and S. T. Chanson, editors, *Protocol Specification, Testing and Verification XIV*, pages 5–20. Chapman and Hall, 1995. IFIP WG 6.1 Symposium.

[Hoa67] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:516–580, 1967.

[Hol89] S. Holström. A refinement calculus for specifications in Henessy-Milner logic with recursion. *Formal Aspects of Computing*, 1:242–272, 1989.

[HU97] I. Hayes and M. Utting. A sequential real-time refinement calculus. Technical Report UQ-SVRC-97-33, Software Verification Research Centre, University of Queensland, 1997. URL http://www.svrc.it.uq.edu.au/.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994. Also appeared as DEC SRC Research Report 79.

[MK93] Y. Moses and O. Kislev. Knowledge-oriented programming. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 261–270, 1993.

[MM98] R. van der Meyden and Y. Moses. Top-down considerations on distributed systems. In *Proc. 12th Int. Symp. on Distributed Computing, DISC'98*, pages 16–19, Andros, Greece, Sept 1998. Springer LNCS No. 1499.

[Mor87] J. M. Morris. A theoretical basis for refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice Hall, New York, 1990.

[UF96] M. Utting and C. Fidge. A real-time refinement calculus that changes only time. In He Jifeng, editor, *Proc. 7th BCS/FACS Refinement Workshop*, Electronic Workshops in Computing. Springer, 1996.

[UF97] M. Utting and C. Fidge. Refinement of infeasible real-time programs. In *Proc. Formal Methods Pacific '97*, Series in Discrete Mathematics and Theoretical Computer Science, pages 243–262, 1997.

[Win86] G. Winskel. A complete proof system for SCSS with modal assertions. *Fundamenta Informaticae*, IX:401–419, 1986.