Too appear in the proceedings of FOSSACS 2000, Foundations of Software Science and Computation Structures, March 27-31, 2000, Berlin, Germany, LNCS ????, ©Springer-Verlag, 2000

A Program Refinement Framework Supporting Reasoning about Knowledge and Time (Preliminary Report)

Kai Engelhardt   Ron van der Meyden*   Yoram Moses†

January 20, 2000

Abstract

This paper develops a highly expressive semantic framework for program refinement that supports both temporal reasoning and reasoning about the knowledge of a single agent. The framework generalizes a previously developed temporal refinement framework by amalgamating it with a logic of quantified local propositions, a generalization of the logic of knowledge. The combined framework provides a formal setting for development of knowledge-based programs, and addresses two problems of existing theories of such programs: lack of compositionality and the fact that such programs often have only implementations of high computational complexity. Use of the framework is illustrated by a control theoretic example concerning a robot operating with an imprecise position sensor.

1 Introduction

The *knowledge-based* approach to the design and analysis of distributed systems, introduced by Halpern and Moses [6] involves the use of modal logics of knowledge. One of the key contributions of this approach is the notion of *knowledge-based programs* [5, 4], which generalize standard programs by allowing the tests in conditional constructs to be formulas in the logic of knowledge. Such programs contain statements of the form "if you know that X then do A else B". This provides a high level abstraction of distributed programs that allows for perspicuous descriptions of how an agent's actions are related to its state of information (which, in a distributed system, is typically incomplete) about its environment.

In its current state of development, the knowledge-based approach has a number of limitations, among them that:

1. The formal methodology for developing and reasoning about knowledge-based programs is at present only weakly developed.
2. The existing semantics for knowledge-based programs is based on a particular interpretation of knowledge that requires a complete description of the implementing program. This prevents the compositional development of program fragments.
3. Knowledge-based programs often have only implementations of unacceptably high computational complexity.

This paper is a step in the direction of the formulation of the knowledge-based approach that addresses these limitations.

One of the starting points for our work is the observation that knowledge-based programs are in one respect more like specifications than like standard programs. They cannot be directly executed — instead, their meaning is defined by a relation of "implementation" between knowledge based programs and standard programs: a given knowledge-based program may have no, one, or many concrete programs as its

*School of Computer Science and Engineering, The University of New South Wales, Sydney 2052 Australia, for $X \in \{\text{kaie}, \text{meyden}\}$, email: X@cse.unsw.edu.au, www: http://www.cse.unsw.edu.au/~X/

†Department of Electrical Engineering, Technion, Haifa, Israel, moses@ee.technion.ac.il

implementations. As a specification formalism, however, knowledge-based programs are unbalanced, abstracting only the tests performed by agents, but providing no abstraction mechanism for their actions [11].

Action abstraction is handled much better in *refinement calculi* [1, 9, 10], also known as "broad spectrum" languages. Such calculi view programs and specifications as having the same semantic type, and support a formal methodology for the development of programs that are "correct by design", where one begins with a specification and transforms it to an implementation by means of a sequence of correctness preserving refinement steps. The focus in this area has been on sequential programs and atemporal assertions but recently some approaches to refinement admitting the expressive power of temporal logics have been developed [14, 7].

A first step in the direction of a refinement calculus suited to the knowledge-based development of programs was taken in van der Meyden and Moses [17, 16], where it is shown how to develop a refinement approach capturing certain types of temporal reasoning that will be critical in knowledge-based program development. We further develop these ideas in the present paper, by showing how they may be extended to accommodate knowledge-based reasoning. Significantly, the framework we define admits compositional program development.

In developing the extension, we also seek to address the final limitation of knowledge-based programs alluded to above. To implement the statement "if you know that X then do A else B", a concrete program must do A exactly when it is in a local state (captured by the values of the variables and storage it maintains locally) that carries the information that X is true. The difficulty with this is that computing whether a local state bears the information that X may have very high computational complexity [12, 15, 18]. As argued by Sanders [13] and us [3], in practice, it may often be sufficient to use conditions on the agent's state of information that are sound, but not complete, tests of its knowledge. Such tests may be expressed in the *Logic of Local Propositions* (LLP) [3].

The present paper integrates the temporal refinement framework of van der Meyden and Moses [16] with the logic of local propositions. Although our ultimate aim is a framework for the development of distributed systems, we deal in this paper with a single agent operating synchronously with its environment: asynchrony and multiple agents introduce complexities that we plan to address in the future. The main novelty is the introduction of a programming/specification construct that resembles a quantification over local propositions. This construct makes it possible to write specifications stating that the agent conditions its behaviour on a *local* test for some property of interest, without stating explicitly what test is used. The introduction of this construct necessitates an adaptation of the semantics of the temporal refinement of [16].

The paper is structured as follows. Section 2 defines an assertion language that adapts the LLP semantics to the richer temporal setting required for reasoning about programs. Section 3 defines the syntax and semantics of our broad spectrum programming and specification language that incorporates the assertion language from Sect. 2. Section 4 defines the semantic refinement relation we use for this class of programs and develops a number of refinement rules valid for this relation. Section 5 illustrates the use of the framework by presenting a formal development of a control theoretic example previously treated informally in the literature on knowledge-based programs.

2 A Semantics for Reasoning about Knowledge and Time

We begin by presenting a semantic framework for a single agent and its environment, inspired by [4], to which we refer the reader for motivation.

Let L_e be a set of possible states for the environment and let L_1 be a set of possible local states for agent 1. We take $\mathcal{G} = L_e \times L_1$ to be the set of *global states*. Let A_1 and A_e be nonvoid sets of *actions* for agent 1 and for the environment, respectively. (These sets usually contain a special *null action* Λ.) A *joint action* is a pair $(a_e, a_1) \in \mathcal{A} = A_e \times A_1$. A *run* over $\mathcal{G}$ and $\mathcal{A}$ is a pair $r = (h, \alpha)$ of infinite sequences: a *state history* $h : \mathbb{N} \longrightarrow \mathcal{G}$, and an *action history* $\alpha : \mathbb{N} \longrightarrow \mathcal{A}$. Intuitively, for $c \in \mathbb{N}$, $h(c)$ is the global state of the system at time c and $\alpha(c)$ is the joint action occurring at time c. (We say more about the transition relation connecting states and actions later.) A *system* over $\mathcal{G}$ and $\mathcal{A}$ is a set of runs over $\mathcal{G}$ and $\mathcal{A}$, intuitively representing all possible histories. A pair (r, c) consisting of a run r (in system S) and a time $c \in \mathbb{N}$ is called a *point (in S)*. We write $\mathrm{Points}(S)$ for the set of points of S. Let *Prop* be a set of propositional variables. An *interpretation* of a system S is a mapping $\pi : Prop \longrightarrow 2^{\mathrm{Points}(S)}$ associating a set of points with each propositional variable.

Intuitively, proposition $p \in Prop$ is true exactly at the points contained in $\pi(p)$. An *interpreted system (over* $\mathcal{G}$ *and* $\mathcal{A}$*)* is a pair $\mathfrak{I} = (S, \pi)$ where S is a system over $\mathcal{G}$ and $\mathcal{A}$ and π is an interpretation of S.

The structure in the above definitions supports the following notions used to define the agent's knowledge. We say two points $(r,c),(r',c')$ in a system S are 1-*indistinguishable*, denoted $(r,c) \sim_1 (r',c')$, if the local components of the global states at these points are equal, i.e., if there exists a local state $s_1 \in L_1$ and states of the environment s_e, s'_e such that $h(c) = (s_e, s_1)$ and $h'(c') = (s'_e, s_1)$, where $r = (h,\alpha)$ and $r' = (h',\alpha')$. A set P of points of S is 1-*local* if it is closed under $\sim_1$, in other words, when for all points $(r,c),(r',c')$ of S, if $(r,c) \in P$ and $(r,c) \sim_1 (r',c')$ then $(r',c') \in P$. Intuitively, 1-local sets of points correspond to properties that the agent is able to determine entirely on the basis of its local state. If π and π' are interpretations and $p \in Prop$, then π' is said to be a 1-*local p-variant* of π, denoted $\pi \simeq^1_p \pi'$, if π and π' differ at most in the value of p and $\pi'(p)$ is 1-local. If $\mathfrak{I} = (S,\pi)$ and $\mathfrak{I} = (S',\pi')$ are two interpreted systems over $\mathcal{G}$ and $\mathcal{A}$, then $\mathfrak{I}'$ is said to be 1-*local p-variant* of $\mathfrak{I}$, denoted $\mathfrak{I} \simeq^1_p \mathfrak{I}'$, if $S = S'$ and $\pi \simeq^1_p \pi'$.

The logical language $\mathcal{L}$ we use in this paper resembles a restricted monadic second order logic with two additions: (a) an S5-modality for necessity and (b) operators from the linear time temporal logic LTL [8]. Its syntax is given by:

$$\mathcal{L} \ni \phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{Nec}\,\phi \mid \forall_1 p\,(\phi) \mid \bigcirc\phi \mid \phi\,\mathsf{U}\,\phi \mid \ominus\phi \mid \phi\,\mathsf{S}\,\phi$$

where $p \in Prop$. Intuitively, $\mathsf{Nec}\,\phi$ says that ϕ is true at all points in the interpreted system, and its dual $\mathsf{Poss}\,\phi = \neg\,\mathsf{Nec}\,\neg\phi$ states that ϕ is true at some point. The formula $\forall_1 p\,(\phi)$ says that ϕ is true for all assignments of a 1-local proposition (set of points) to the propositional variable p. We write $\exists_1 p\,(\phi)$ for its dual $\neg\forall_1 p\,(\neg\phi)$. The remaining connectives have their standard interpretations from linear time temporal logic: $\bigcirc$ ("next"), U ("until"), $\ominus$ ("previously") and S ("since"). We employ parenthesis to indicate aggregation and use standard abbreviations such as *true*, *false*, $\vee$, and definable future time operators like $\square$ ("henceforth") and $\diamondsuit$ ("eventually"), as well as their past time counterparts $\boxminus$ ("until now") and $\diamondminus$ ("once").

Formulae of $\mathcal{L}$ are interpreted at a point (r,c) of an interpreted system $\mathfrak{I} = (S,\pi)$ by means of the satisfaction relation $\models$, defined inductively by:

- $\mathfrak{I},(r,c) \models p$ iff $(r,c) \in \pi(p)$;
- $\mathfrak{I},(r,c) \models \neg\phi$ iff $\mathfrak{I},(r,c) \not\models \phi$;
- $\mathfrak{I},(r,c) \models \phi \wedge \psi$ iff $\mathfrak{I},(r,c) \models \phi$ and $\mathfrak{I},(r,c) \models \psi$;
- $\mathfrak{I},(r,c) \models \mathsf{Nec}\,\phi$ iff $\mathfrak{I},(r',c') \models \phi$, for all $(r',c') \in \mathrm{Points}(S)$;
- $\mathfrak{I},(r,c) \models \forall_1 p\,(\phi)$ iff $\mathfrak{I}',(r,c) \models \phi$ for all $\mathfrak{I}'$ such that $\mathfrak{I} \simeq^1_p \mathfrak{I}'$;
- $\mathfrak{I},(r,c) \models \bigcirc\phi$ iff $\mathfrak{I},(r,c+1) \models \phi$;
- $\mathfrak{I},(r,c) \models \phi\,\mathsf{U}\,\psi$ iff there exists a $d \geq c$ such that $\mathfrak{I},(r,d) \models \psi$ and $\mathfrak{I},(r,e) \models \phi$ for all e with $c \leq e < d$;
- $\mathfrak{I},(r,c) \models \ominus\phi$ iff $c > 0$ and $\mathfrak{I},(r,c-1) \models \phi$;
- $\mathfrak{I},(r,c) \models \phi\,\mathsf{S}\,\psi$ iff there exists a $d \leq c$ such that $\mathfrak{I},(r,d) \models \psi$ and $\mathfrak{I},(r,e) \models \phi$ for all e with $d < e \leq c$.

Given these constructs, it is possible to express many operators from the literature on reasoning about knowledge. For example, consider the standard knowledge operator K_1, defined by $\mathfrak{I},(r,c) \models K_1\phi$ if $\mathfrak{I},(r',c') \models \phi$ for all points (r',c') of $\mathfrak{I}$ such that $(r,c) \sim_1 (r',c')$. This is expressible as $\exists_1 p\,(p \wedge \mathsf{Nec}(p \rightarrow \phi))$. We refer to [3] for further examples and discussion.

3 Sequential Programs with Quantification over Local Propositions

In this section we define our wide spectrum programming language, and discuss its semantics. We also define a refinement relation on programs.

3.1 Syntax

The programming language describes the structure of segments of runs. Let CV be a set of *constraint variables* and PV a set of *program variables*. Define the syntactic category Prg of *programs* by

$$Prg \ni P ::= \varepsilon \mid Z \mid a \mid P * P \mid P + P \mid P^{\omega} \mid \exists_1 p\,(P) \mid [\phi,\psi]^X \mid [\phi]^X \mid \{\phi\}_C$$

where $Z \in PV$, $a \in A_1$, $p \in Prop$, $\phi,\psi \in \mathcal{L}$, $X \in CV$, and $C \subseteq CV$. The intuitive meaning of these constructs is as follows. The symbol ε denotes the *empty program*, which takes no time to execute, and has no effects. Program variables Z are placeholders used to allow substitution of programs. Note that a program may refer directly to actions a of the agent, but the actions of the environment are left implicit. The operation $*$ represents *sequential composition*. The symbol $+$ denotes nondeterministic choice, while P^{ω} denotes zero or more (possibly infinitely many) repetitions of P. The construct $\exists_1 p\,(P)$ can also be understood as a kind of nondeterministic choice: it states that P runs with respect to some assignment of a 1-local proposition to the propositional variable p. The last three constructs are like certain constructs found in refinement calculi. Intuitively, the *specification* $[\phi,\psi]^X$ states that some program runs in this location that has the property that, if started at a point satisfying ϕ, eventually terminates at a point satisfying ψ.[1] The *coercion* $[\phi]^X$ is a program that takes no time to execute, but expresses a constraint on the surrounding program context: this must guarantee that ϕ holds at this location. The constraint variable X in specifications and coercions acts as a label that allows references by other pieces of program text. Specifically, this is done in the *assertions* $\{\phi\}_C$, which act like program annotations: such a statement takes no time to execute, and, intuitively, asserts that ϕ can be proved to hold at this program location, with the proof depending only on concrete program fragments and on specification and coercion statements whose labels are in C. We may omit the constraint variables when it is not necessary to make such references.

In programs "$*$" binds tighter than "$+$". We employ parentheses to indicate aggregation wherever necessary and tend to omit $*$ near coercions and assertions. Moreover, we use the following abbreviations: **if**^{X} ϕ **then** P **else** Q **fi** $= [\phi]^X P + [\neg\phi]^X Q$ and **while**^{X} ϕ **do** P **od** $= ([\phi]^X P)^{\omega}[\neg\phi]^X$. Our programming language can express some programs closely related to the knowledge-based programs of [4]. These are program such as:

case of
  if $K_1\phi$ **do** a_1
  if $\neg K_1\psi$ **do** a_2
end case

A program closely related to this is $([K_1\phi]\,a_1 + [\neg K_1\psi]\,a_2 + [\neg(K_1\phi \vee \neg K_1\psi)]\,\Lambda)^{\omega}$. The precise relationship is subtle and deferred to the full version of this paper.

3.2 Semantics

Our semantics will treat programs like specifications of certain sets of run segments in a system, intuitively, the sets of run segments that can be viewed as having been generated by executing the program. We note that the semantics presented in this section treats assertions $\{\phi\}_C$ as equivalent to the null program ε — the role of assertions in the framework will be explained later.

We first define execution trees, which represent unfoldings of the nondeterminism in a program. It is convenient to represent these trees as follows. A *binary tree domain* is a prefix-closed subset of the set $\{0,1\}^* \cup \{0,1\}^{\omega}$. So, each nonvoid tree domain contains the empty sequence λ. Let A be a set. An *A-labelled binary tree* is a function T from a binary tree domain D to A. The *nodes* of T are the elements of D. The node λ is called the *root* of T. If $n \in D$ we call $T(n)$ the *label* at node n. If $n \in D$ then the *children of n in T* are the nodes of T (if any) of the form $n \cdot i$ where $i \in \{0,1\}$. Finite maxima in the prefix order on D are called *leaves* of T.

An *execution tree* is a Prg-labelled binary tree, subject to the following constraints on the nodes n:

[1] In refinement calculi, such statements are typically associated with *frame variables*, representing the variables allowed to change during the execution — we could add these, but omit them for brevity.

1. If n is labelled by ε, a program variable $Z \in PV$, a basic action a, a specification $[\phi,\psi]^X$, a coercion $[\phi]^X$, or an assertion $\{\phi\}_C$, then n is a leaf.

2. If n is labelled by $\exists_1 p\,(P)$ then n has exactly one child $n \cdot 0$, labelled by P.

3. If n is labelled by $P * Q$ or $P + Q$ then n has exactly two children $n \cdot 0$, $n \cdot 1$, labelled by P and Q respectively.

4. If n is labelled by P^ω then n has exactly two children, $n \cdot 0$, $n \cdot 1$, labelled by ε and $P * (P^\omega)$, respectively.

With each program P we associate a particular execution tree, T_P, namely the unique execution tree labelled with P at the root λ.

We now define the semantic constructs specified by programs. An *interval in a system S* is a triple $r[c,d]$ consisting of a run r of S and two elements c and d of $\mathbb{N}_+ = \mathbb{N} \cup \{\infty\}$ such that $c \leq d$. We say that the interval is *finite* if $d < \infty$. A set I of intervals is *run-unique* if $r[c,d], r[c',d'] \in I$ implies $c = c'$ and $d = d'$. An *interpreted interval set over S* (or *iis* for short) is a pair (π, I) consisting of an interpretation π of S and a run-unique set I of intervals over S.

We will view programs as specifying, or executing over, interpreted interval sets, by means of certain mappings from execution trees to interpreted interval sets. To facilitate the definition in the case of sequential composition, we introduce a shorthand for the two sets obtained by splitting each interval in a given set I of intervals of S in two. Say that $f : I \longrightarrow \mathbb{N}_+$ *divides I* whenever $c \leq f(r[c,d]) \leq d$ holds for all $r[c,d] \in I$. Given some f dividing I, we write $f_\blacktriangleleft(I)$ for the set of intervals $r[f(r[c,d]),d]$ such that $r[c,d] \in I$. Analogously, we write $f_\blacktriangleright(I)$ for $\{\, r[c, f(r[c,d])] \mid r[c,d] \in I \,\}$.

Let S be a system, let (π, I) be an iis w.r.t. S, and let P be a program. A function θ mapping each node n of T_P to an iis $(\pi_\theta(n), I_\theta(n))$, respectively, is an *embedding* of T_P in (π, I) w.r.t. S whenever the following conditions are satisfied:

1. $\theta(\lambda) = (\pi, I)$.

2. If n is labelled ε or $\{\phi\}_C$, then $c = d$ for all $r[c,d] \in I_\theta(n)$.

3. If n is labelled a then, for all $(h,\alpha)[c,d] \in I_\theta(n)$, if $c < \infty$ then both $d = 1 + c$ and $a = a_1$, where $\alpha(c) = (a_e, a_1)$.

4. If n is labelled $[\phi,\psi]$, then, for all $r[c,d] \in I_\theta(n)$, whenever $c < \infty$ and $(S, \pi_\theta(n)), (r,c) \models \phi$, then both $d < \infty$ and $(S, \pi_\theta(n)), (r,d) \models \psi$.

5. If n is labelled $[\phi]$, then $c < \infty$ implies that $c = d$ and $(S, \pi_\theta(n)), (r,c) \models \phi$, for all $r[c,d] \in I_\theta(n)$.

6. If n is labelled $\exists_1 p\,(Q)$ then $\pi_\theta(n) \simeq^1_p \pi_\theta(n \cdot 0)$ and $I_\theta(n \cdot 0) = I_\theta(n)$.

7. If n is labelled $Q_1 + Q_2$, then $\pi_\theta(n \cdot 0) = \pi_\theta(n \cdot 1) = \pi_\theta(n)$ and $I_\theta(n)$ is the disjoint union of $I_\theta(n \cdot 0)$ and $I_\theta(n \cdot 1)$.

8. If n is labelled $Q_1 * Q_2$, then $\pi_\theta(n \cdot 0) = \pi_\theta(n \cdot 1) = \pi_\theta(n)$ and there is an f dividing $I_\theta(n)$ such that $I_\theta(n \cdot 0) = f_\blacktriangleright(I_\theta(n))$ and $I_\theta(n \cdot 1) = f_\blacktriangleleft(I_\theta(n))$.

9. If n is labelled Q^ω then $\pi_\theta(n \cdot 0) = \pi_\theta(n \cdot 1) = \pi_\theta(n)$ and $I_\theta(n)$ is the disjoint union of $I_\theta(n \cdot 0)$ and $I_\theta(n \cdot 1)$ (as in case 7) and, for all $r[c,d] \in I_\theta(n)$:
   $d = \bigsqcup \{\, d' \mid r[c',d'] \in I_\theta(n \cdot m)$ for some leaf $n \cdot m$ of T_P below $n \,\}$.

We write $S, (\pi, I) \Vdash_\theta P$ whenever θ is an embedding of T_P in (π, I) w.r.t. S. Say that P *occurs over* (π, I) *w.r.t. S* if there exists a θ such that $S, (\pi, I) \Vdash_\theta P$.

Clauses 1 to 8 formalize the intuitive understanding given above for each of the program constructs. Concerning clause 9 of this definition, we remark that, by run-uniqueness and the other clauses, if $n \cdot m_0, n \cdot m_1 \ldots$ are the leaves $n \cdot m$ below n for which $I_\theta(n \cdot m)$ contains an interval on r, in left to right order, and these intervals are $r[c_0,d_0], r[c_1,d_1], \ldots$, respectively, then we have $d_i = c_{i+1}$ for each index i in the sequence. (We may have $c_i = d_i$.) Intuitively, if d were not the least upper bound d' of the d_i, then this sequence of intervals would amount to an execution of Q^ω over $r[c,d']$ rather than over $r[c,d]$. (See [16] for further motivation.)

3.3 Refinement

The semantics just presented can be shown to be a generalization of the semantics of [16] for a similar language without the local propositional quantifier. That semantics, however, dealt with *single* intervals where we have used a set of intervals. The motivation for the change is that certain undesirable refinement rules involving the local propositional quantifier would be valid under the earlier semantic approach. We now present two definitions of refinement and an example that motivates the richer semantics.

Intuitively, a program P refines Q if, whenever P executes, so does Q. A refinement relation of this type, when transitive and preserved under program composition, allows us to start with a high level specification and derive a concrete implementation through a sequence of refinement steps.

One refinement relation definable using our semantics as is follows: P refines Q, denoted $P \sqsubseteq Q$ when for all systems S, and interpreted interval sets (π, I) over S, if $S,(\pi,I) \Vdash P$ then $S,(\pi,I) \Vdash Q$. For the semantics using single intervals, the corresponding relation would be defined by $P \sqsubseteq^* Q$ when for all systems S, interpretations π and intervals $r[c,d]$ of S, if $S,(\pi,\{r[c,d]\}) \Vdash P$ then $S,(\pi,\{r[c,d]\}) \Vdash Q$. Clearly, if $P \sqsubseteq Q$ then $P \sqsubseteq^* Q$. As the following example demonstrates, the converse is false.

Example 3.1 Let $\phi \in \mathcal{L}$ be any formula and consider the following two programs.

$$P = \textbf{if}\ \phi\ \textbf{then}\ a\ \textbf{else}\ a * a\ \textbf{fi} \qquad\qquad Q = \exists_1 p\,(\textbf{if}\ p\ \textbf{then}\ a\ \textbf{else}\ a * a\ \textbf{fi})$$

We shall first show that $P \sqsubseteq^* Q$ and then argue that this is not desirable. Suppose $S,(\pi,\{r[c,d]\}) \Vdash P$. Recall that an **if** statement abbreviates a nondeterministic choice. Thus, there are two cases to be considered:

Case 1: $S,(\pi,\{r[c,d]\}) \Vdash [\phi]\,a$. Define the 1-local p-variant π' of π by $\pi'(p) = \text{Points}(S)$, that is, p is everywhere true under π'. It follows that $S,(\pi',\{r[c,d]\}) \Vdash [p]\,a$, and thus, $S,(\pi',\{r[c,d]\}) \Vdash$ **if** p **then** a **else** $a*a$ **fi**. By definition, $S,(\pi,\{r[c,d]\}) \Vdash Q$.

Case 2: $S,(\pi,\{r[c,d]\}) \Vdash [\neg\phi]\,a*a$. This is handled analogously by defining $\pi'(p) = \emptyset$.

To see that it is not the case that $P \sqsubseteq Q$, take ϕ to be a propositional variable q. It is straightforward to construct a system S, finite intervals $i = r[c,d]$ and $i' = r'[c',d']$, and interpretation π such that $S,(\pi,\{i\}) \Vdash [q]\,a$ and $S,(\pi,\{i'\}) \Vdash [\neg q]\,a*a$. Hence $S,(\pi,\{i,i'\}) \Vdash$ **if** q **then** a **else** $a*a$ **fi**), but (r,c) and (r',c') are 1-indistinguishable. If we were to have $S,(\pi,\{i,i'\}) \Vdash \exists_1 p\,($**if** p **then** a **else** $a*a$ **fi**), then we would have a 1-local p-variant π' of π such that $S,(\pi',\{i,i'\}) \Vdash$ **if** p **then** a **else** $a*a$ **fi**. But by assumption $(r,c) \in \pi'(p)$ iff $(r',c') \in \pi'(p)$, so we have either $S,(\pi',\{i,i'\}) \Vdash a$ or $S,(\pi',\{i,i'\}) \Vdash a*a$. But neither of these is possible, since one or the other interval has the wrong length.

Our intuition in writing Q is that it specifies a program that chooses to do either a or $a*a$ on the basis of some locally computable test p. The refinement $P \sqsubseteq^* Q$ is contrary to this intuition: it states that Q may be implemented by using in place of p *any* test, even one not locally computable. Intuitively, this result is obtained by using a different 1-local test in different executions of the program. Our semantics has been designed so as to avoid this: it ensures that a *uniform* test p is used in every execution of the program. Thereby, the undesirable refinement is blocked.

We remark that a slight variant of the example is a valid, and desired refinement: $[\exists_1 p\,(\mathsf{Nec}(p \equiv \phi))]\,P \sqsubseteq Q$. Here, the coercion states that ϕ is in fact equivalent to a 1-local proposition. We will use this rule below. ■

4 Validity and Valid Refinement

We now briefly discuss the role of assertions $\{\phi\}_C$ in the framework and define the associated semantic notions. The reader is referred to [16] for a more detailed explanation of these ideas in a simpler setting.

Intuitively, an assertion $\{\phi\}_C$ is like an annotation at a program location stating that ϕ is guaranteed to hold whenever the program execution reaches this location. Moreover, such an assertion states that this fact "depends" only on constraints in the program (specifications and coercions) labelled with constraint variables in the set C, as well as on concrete program fragments. (We do not include labels for these because they cannot be "refined away".) The reason we include the justification C for the assertion is that it proves to be necessary to track such information in order to be able to formulate a number of desirable refinement rules. These rules refine a program fragment in ways that depend upon the larger program context within which the fragment occurs.

One typical example of this is a rule concerning the elimination of coercions. Suppose a coercion $[\phi]$ occurs at a program location where ϕ is guaranteed to hold. Intuitively, we would like to say that the coercion can be eliminated (replaced by ε) in such circumstances. However, the attempt to formulate this by the refinement rule $\varepsilon \leq \{\phi\}\,[\phi]$ is not quite correct, for the reason the assertion holds could be the very coercion we seek to eliminate. (It may seem a little odd at first to say that the justification for the assertion is some part of the program text that follows, but consider the case of $\phi = \Diamond\psi$. See [16] for an example that makes essential use of assertions justified by later pieces of program text.) The use of justifications enables us to formulate the rule as $\varepsilon \leq \{\phi\}_C\,[\phi]^X$, *provided* X is not in C, i.e., provided the assertion does not rely upon the coercion. This blocks the circular reasoning.

The semantics of assertions is formalized as follows. In order to capture constraint dependencies, we first define for each program P and constraint set $C \subseteq CV$ a program $\mathrm{relax}(P,C)$ that is like P, except that only constraints whose labels are in C are enforced: all other constraints are relaxed. Formally, we obtain $\mathrm{relax}(P,C)$ from P by replacing each occurrence of a coercion $[\phi]^X$ where $X \notin C$ by ε, and also replacing each occurrence of a specification $[\phi,\psi]^X$ where $X \notin C$ by $[\mathit{false},\mathit{true}]^X$ in P^C.

We may now define a program P to be *valid* with respect to a set of interpreted systems $\mathcal{S}$ when for all assertions $\{\phi\}_C$ in P, all interpreted systems $(S,\pi) \in \mathcal{S}$ and all intervals sets I over S, all embeddings θ of $T_{\mathrm{relax}(P,C)}$ into $S,(I,\pi)$ have the property that for all nodes n of $T_{\mathrm{relax}(P,C)}$ labelled with $\{\phi\}_C$, we have $S,\theta(n) \Vdash [\phi]$. Intuitively, the embedding represents an execution of P in which only constraints in C are enforced, and we check that the associated assertions hold at the appropriate points in the execution. Note that when n is labelled by an assertion, $I_\theta(n)$ must be a set of intervals of length 0. Moreover, the semantics of $S,(I,\pi) \Vdash [\phi]$ checks ϕ only at finite points in this set. Thus, validity can be understood as a kind of generalized partial correctness. We define validity with respect to a set of interpreted systems $\mathcal{S}$ to allow assumptions concerning the environment to be modelled: e.g., $\mathcal{S}$ might be the set of all interpreted systems in which actions have specific intended interpretations. We give an example of this in the next section.

Clearly, we want to avoid programs that are not valid (such as $[p]^X\,\{\neg p\}_{\{X\}}$). Thus, we would now like a notion of refinement that preserves validity, so that we derive only valid programs from valid programs by refinement. The refinement relation $\sqsubseteq$ defined above does not have this property. However, we may use it to define a notion that does. In order to do so, we first need to define a technical notion. A *justification transformation* is a mapping $\eta : 2^{CV} \longrightarrow 2^{CV}$ that is increasing, i.e., satisfies $C \subseteq \eta(C)$ for all $C \subseteq CV$. The result of applying a justification transformation η to a program P is the program $P\eta$ obtained by replacing each instance of an assertion $\{\phi\}_C$ in P by the assertion $\{\phi\}_{\eta(C)}$. When $R(Z)$ is a program containing a program variable Z and P is a program, we write $R\eta(P)$ for the result of first applying η to $R(Z)$ and then substituting P for Z. We need such transformations for refinements such as replacing $\{\phi\}_C[\phi]^X$ by ε when $X \notin C$ within some large program context. Intuitively, when we do this, any assertion in the larger context that depended on the coercion labelled X is still valid, but its justification should now include C in place of X.

The identity justification transformation is denoted by ι. We will also represent justification transformations using expressions of the form $X \hookrightarrow D$, where $X \in CV$ and $D \subseteq CV$. Such an expression denotes the justification transformation η such that $\eta(C) = C \cup D$ if $X \in C$ and $\eta(C) = C$ otherwise.

Let $\mathcal{S}$ be a set of interpreted systems, let η be a justification transformation and let P and Q be programs. Say that *P validly refines Q in $\mathcal{S}$ under* η, and write $P \leq^{\mathcal{S}}_{\eta} Q$, if for all programs $R(Z)$ with Z a program variable, if $R(Q)$ is valid with respect to $\mathcal{S}$ then $R\eta(P)$ is valid with respect to $\mathcal{S}$, and for all $(S,\pi) \in \mathcal{S}$ and interval sets I over S, if $S,(I,\pi) \Vdash R\eta(P)$ then $S,(I,\pi) \Vdash R(Q)$.

We remark that other definitions of valid refinement are possible. While intuitive, the definition above is very sensitive to the syntax of the programming language. We will consider some closely related semantic alternatives elsewhere.

4.1 Valid Refinement Rules

We now present a number of rules concerning valid refinement that are sound with respect to the semantics just presented, making no attempt at completeness. We focus on rules concerning the existential quantifiers, and refer to [16] for additional rules concerning the other constructs, which are also sound in the framework of the present paper.

The following rules make it possible for refinement to broken down into a sequence of steps that operate on small program fragments. (Only justification transformation operate globally, but this can also be managed locally by means of appropriate data structures.)

$$\frac{P \leq^{\mathbb{S}}_{\eta} Q,\, Q \leq^{\mathbb{S}}_{\eta'} R}{P \leq^{\mathbb{S}}_{\eta\circ\eta'} R} \qquad\qquad \frac{P \leq^{\mathbb{S}}_{\eta} Q}{R\eta(P) \leq^{\mathbb{S}}_{\eta} R(Q)}$$

Reducing the amount of nondeterminism and introducing a coercion are sound refinement steps.

$$P \leq^{\mathbb{S}}_{\iota} P + Q \qquad\qquad [\phi] \leq^{\mathbb{S}}_{\iota} \varepsilon$$

Quantification over local propositional variables can be introduced, extracted from a coercion, and lifted to contexts.

$$\exists_1 p\,(P) \leq^{\mathbb{S}}_{\iota} P \quad \textit{if } p \textit{ not free in } P \qquad \textbf{i-lq}$$

$$\exists_1 p\,([\phi]) \leq^{\mathbb{S}}_{\iota} [\exists_1 p\,(\phi)] \qquad \textbf{ext-lq}$$

$$\exists_1 p\,(R(P)) \leq^{\mathbb{S}}_{\iota} R(\exists_1 p\,(P)) \quad \textit{if } p \textit{ not free in } R(Z) \qquad \textbf{lift-lq}$$

Let P_ϕ denote the program obtained from P by substituting formula ϕ for all free occurrences of p in P, while taking the usual care of free variables in ϕ by renaming clashing bound variables in P.

$$[\exists_1 p\,(\mathsf{Nec}(\phi \equiv p))]\, P_\phi \leq^{\mathbb{S}}_{\iota} \exists_1 p\,(P) \qquad \textbf{inst-lp}$$

4.2 Single-Stepping Programs and Loops

Reasoning about termination of a loop, say, **while** g **do** P **od** becomes easier when strict bounds on the running time of P are known. We present here a simple example of this phenomenon that is useful for the example we present in Sect. 5. More general rules can be formulated than the one we develop here.

Say that program P is *single-stepping*, if $S,(\pi,I) \Vdash P$ and $r[c,d] \in I$ and $c < \infty$ imply that $d = 1 + c$, for all S, π, and I. In a slightly broader syntax with existential quantification over arbitrary propositions, not just local ones, the fact that P is single-stepping could be expressed by:

$$P \leq^{\mathbb{S}}_{\eta} \exists p\,([\bigcirc\,\mathsf{first}\,p]\,[\mathit{true}, \mathsf{first}\,p]) \quad .$$

where $\mathsf{first}\,\phi$ is an abbreviation for $\phi \wedge \neg\circleddash\Diamond\phi$, which holds exactly at the first point in a run that makes ϕ true. This notion can be combined with the usual pre/post-condition style of specifying P's behaviour to specify that P is single-stepping and terminates in points satisfying ψ when started in points satisfying ϕ:

$$P \leq^{\mathbb{S}}_{\eta} \exists p\left([\bigcirc\,\mathsf{first}\,p]^X\,[\mathit{true}, \mathsf{first}\,p \wedge (\circleddash\phi \to \psi)]^X\right)$$

Denote the RHS of the above by $\mathbf{ss}[\phi,\psi]^X$. So $S,(\pi,I) \Vdash \mathbf{ss}[\phi,\psi]^X$ if for all $r[c,d] \in I$, whenever $c < \infty$ and $(S,\pi),(r,c) \models \phi$, then both $d = c+1$ and $(S,\pi),(r,d) \models \psi$. Observe that $\mathbf{ss}[\phi,\psi]^X$ takes a single step regardless of whether ϕ holds initially. Consequently, $\mathbf{ss}[\phi,\psi]^X$ is indeed single-stepping. Adding the single-stepping requirement yields a valid refinement: $\mathbf{ss}[\phi,\psi]^X \leq^{\mathbb{S}}_{\iota} [\phi,\psi]^X$. The following rule for single-stepping loop bodies will be used in Section 5.

$$[\phi \to \psi]^X\, \exists_1 p \begin{pmatrix} [\psi \to \Diamond p]^X * \\ \mathbf{while}^X\ \neg p\ \mathbf{do}\ \mathbf{ss}[\psi \wedge \neg p, \psi]^X\ \mathbf{od} * \\ [\psi \wedge p \to \phi']^X \end{pmatrix} \leq^{\mathbb{S}}_{\iota} [\phi,\phi']^X \qquad \textbf{i-ss-loop}$$

To apply this rule, one has to invent a (not necessarily local) loop invariant ψ. Finding a concrete local guard is postponed via use of the existential quantification. Just as for ordinary sequential programs, the first and last coercion link the invariant to the pre- and postcondition of the specification that is to be implemented. The second coercion, $[\psi \to \Diamond p]^X$ ensures termination of the loop.

5 Example: Autonomous Robot

In this section we discuss an example that closely resembles Example 7.2.2 in [4] which in turn has been inspired by the 1994 conference version of [2].

A robot travels along an endless corridor, which in this example is identified with the natural numbers. The robot starts at 0 and has the goal of stopping in the goal region $\{2,3,4\}$. To judge when to stop the robot has a sensor that reads the current position. (See Fig. 1.) Unfortunately, this sensor is inaccurate; the

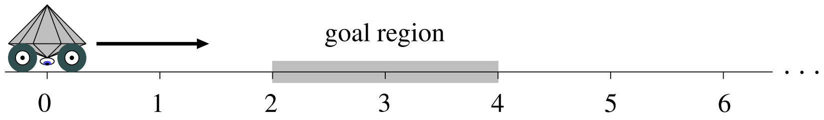

Figure 1: Autonomous Robot

readings may be wrong by at most 1. The only action the robot can actively take is halting, the effect of which is instantaneous stopping. Unless this action is taken, the robot may move by steps of length 1 to higher numbers. Unless it has taken its halting action, it is beyond its control whether it moves in a step. Our task is now to design a control program for the robot such that:
(safety) The robot only stops in the goal region.
(liveness) The robot is guaranteed to stop eventually.
A modest assumption about the environment is needed for the latter to be achievable. We insist that it is not the case that the robot sits still forever without moving forward or taking the halting action.

To model these assumptions we introduce a system constraint reflecting the following conditions. Strictly speaking, our specification language $\mathcal{L}$ only contains variables that are interpreted as Boolean values but none for natural numbers. It is possible to present this example only using propositions by sacrificing legibility. An extension of our framework to typed variables is straightforward and omitted here for brevity. Let $\mathcal{S}$ be the set of interpreted systems satisfying the following constraints.

1. Initially, the robot's position x is zero: $\mathit{init} \to x = 0$, where init abbreviates the formula $\neg\circleddash \mathit{true}$, which holds exactly in the initial points of runs.

2. Proposition h is initially false and it is becomes true once the robot has halted. Halting is an irreversible action ($h \to \bigcirc h$) and means that the robot does not move anymore: $h \to x = \bigcirc x$.

3. Proposition m is true iff the robot moves in the current step. Moving means that the robot's position is increased by one, otherwise it is unchanged: $(m \to x+1 = \bigcirc x) \wedge \neg m \to x = \bigcirc x$.

4. If the robot has not halted it should move eventually: $(\neg h)\,\mathsf{U}\,(h \vee m)$.

5. The robot's sensor reading is s (an integer) and off by at most one from x, the actual position: $x-1 \leq s \leq x+1$.

6. Only the robot's basic action halt immediately halts the robot.

The variables and propositions mentioned in the constraints are *reserved* in the sense that quantification over them is is not allowed. Thus they essentially "behave" the same in each $(S,\pi) \in \mathcal{S}$. In the full paper we introduce a syntactic representation for such system constraints, give a formal semantics, and introduce valid refinement rules that exploit these constraints. These rules fall into two classes: assertion introduction rules and rules for specification implementation by basic actions. A typical assertion introduction rule for this particular $\mathcal{S}$ is

$$\{\mathit{init} \to x = 0 \wedge \neg h\}_{\emptyset} \preceq^{\mathcal{S}}_{\iota} \varepsilon \tag{1}$$

allowing one to assert a property of initial states in interpreted systems contained in $\mathcal{S}$. For the halting action we would have

$$halt \preceq_\iota^{\mathcal{S}} \mathbf{ss}[true, h \wedge x = \ominus x] \quad . \tag{2}$$

For lack of space we have simplified and pruned the set-up to the above. We refer to "use $\mathcal{S}$" instead of formal refinement rules at points of our derivation that refer to the rules omitted.

In [4] a run-based specification of the system is given by a temporal logic formula equivalent to $\Box(h \to g) \wedge \Diamond h$, where g abbreviates being in the goal region, i.e., $2 \le x \le 4$. The two conjuncts respectively formalize the safety and liveness property from above. The main problem in finding the robot's protocol is to derive a suitable local condition for halting.

We formally derive a protocol for the robot from as abstract as possible a specification of the protocol. The point of departure of our derivation below merely states that the robot must eventually halt in the goal region when started in an initial state.

$$
\begin{array}{ll}
 & [init, g \wedge h]^X \\
\succeq_\iota^{\mathcal{S}} & \text{(sequential composition [16])} \\
 & [init, g]^X * [g, g \wedge h]^X \\
\succeq_\iota^{\mathcal{S}} & \text{(use } \mathcal{S} \text{ to establish } halt \preceq_\iota^{\mathcal{S}} [g, g \wedge h]\text{, cf. (2))} \\
 & [init, g \wedge \neg h]^X * halt \\
\succeq_\iota^{\mathcal{S}} & \text{(}\textbf{i-ss-loop}\text{ with loop invariant } x \le 4 \text{ to prevent exiting the goal region)} \qquad \clubsuit \\
 & [init \to x \le 4]^X * \\
 & \exists_1 p \left([x \le 4 \to \Diamond p]^X \, \mathbf{while}^X \neg p \, \mathbf{do} \, \mathbf{ss}[x \le 4 \wedge p, x \le 4]^X \, \mathbf{od} \, [x \le 4 \wedge p \to g]^X \right) * \\
 & halt \\
\succeq_\iota^{\mathcal{S}} & \text{(use } \mathcal{S} \text{ as in (1) to assert } init \to x \le 4\text{, eliminate coercion)} \\
 & \exists_1 p \left([x \le 4 \to \Diamond p]^X \, \mathbf{while}^X \neg p \, \mathbf{do} \, \mathbf{ss}[x \le 4 \wedge p, x \le 4]^X \, \mathbf{od} \, [x \le 4 \wedge p \to g]^X \right) * \\
 & halt
\end{array}
$$

At this point we select the local test p. The need to satisfy coercion $x \le 4 \wedge p \to g$ together with the fact that the sensor reading differs from the position x by at most 1, leads naturally to the choice $p = s > 2$.

$$
\begin{array}{ll}
\succeq_\iota^{\mathcal{S}} & \text{(}\textbf{inst-lp}\text{)} \\
 & [\exists_1 p \, (\mathsf{Nec}(p \equiv (s > 2)))]^Y * [x \le 4 \to \Diamond s > 2]^X * \\
 & \mathbf{while}^X \, s \le 2 \, \mathbf{do} \, \mathbf{ss}[x \le 4 \wedge s \le 2, x \le 4]^X \, \mathbf{od} * [x \le 4 \wedge s > 2 \to g]^X * halt \\
\succeq_\iota^{\mathcal{S}} & \text{(eliminate two coercions using } \mathcal{S}\text{)} \\
 & [x \le 4 \to \Diamond s > 2]^X * \mathbf{while}^X \, s \le 2 \, \mathbf{do} \, \mathbf{ss}[x \le 4 \wedge s \le 2, x \le 4]^X \, \mathbf{od} * halt \\
\succeq_\iota^{\mathcal{S}} & \text{(use } \mathcal{S} \text{ for } \Lambda \preceq_\eta^{\mathcal{S}} \mathbf{ss}[x \le 4 \wedge s \le 2, x \le 4]^X\text{)} \\
 & [x \le 4 \to \Diamond s > 2]^X * \mathbf{while}^X \, s \le 2 \, \mathbf{do} \, \Lambda \, \mathbf{od} * halt \\
\succeq_\iota^{\mathcal{S}} & \text{(introduce coercion and strengthen coercion [16])} \\
 & [init]^Y * [\Diamond s > 2]^X * \mathbf{while}^X \, s \le 2 \, \mathbf{do} \, \Lambda \, \mathbf{od} * halt
\end{array}
$$

The coercion $\Diamond s > 2$ can be eliminated by reasoning about both the program and $\mathcal{S}$. From the initial state predicate it follows that the loop begins in a state satisfying $\neg h$. The only action executed in the loop is Λ, which in $\mathcal{S}$ preserves the value of h. On termination of the loop the guard must be false, i.e., $s > 2$. In (the purely hypothetical) case the loop diverges the run satisfies $\Box \neg h$, which together with point 4, $(\neg h) \, \mathsf{U} \, (h \vee m)$, allows us to conclude that the robot moves infinitely often. But this also implies that eventually $s > 2$.

$$\succeq_\iota^{\mathcal{S}} \qquad \text{(use } \mathcal{S} \text{ and the loop)}$$

$$
\begin{aligned}
& [init]^Y * \{\diamondsuit(s>2)\}_Y * [\diamondsuit(s>2)]^X \,\mathbf{while}^X\ s \le 2\ \mathbf{do}\ \Lambda\ \mathbf{od} * halt \\
\geq^{\mathbb{S}}_{X \hookrightarrow \{Y\}} \quad & \text{(eliminate coercion)} \\
& [init]^Y \,\mathbf{while}^X\ s \le 2\ \mathbf{do}\ \Lambda\ \mathbf{od} * halt
\end{aligned}
$$

Finally, the rule

$$\frac{[\phi]\, P \leq^{\mathbb{S}}_{\eta} [\phi,\psi]^X}{P \leq^{\mathbb{S}}_{\eta} [\phi,\psi]^X}$$

proves $\mathbf{while}^X\ s \le 2\ \mathbf{do}\ \Lambda\ \mathbf{od} * halt \leq^{\mathbb{S}}_{X \hookrightarrow \{Y\}} [init, g \wedge h]^X$, yielding a concrete implementation.

An alternative derivation from point ♣ onwards indicates how the knowledge-based approach could be modeled in our framework. Firstly we would choose just *true* as loop invariant. Secondly, instead of guessing the appropriate local exit condition $s > 2$ we would let the robot execute the loop until it *knows* that it is in the goal region, i.e., instantiate p with $K_1 g$. The derivation then proceeds as before till reaching the stage before eliminating the last coercion concerning completeness of the test:

$$[init]^Y * [\diamondsuit(K_1 g)]^Y \,\mathbf{while}^X\ \neg K_1 g\ \mathbf{do}\ \Lambda\ \mathbf{od} * halt$$

To develop this to an implementation, that is, eliminate $[\diamondsuit(K_1 g)]^Y$, requires additional features to be introduced into the framework, so we will not pursue this here.

6 Conclusion and Future Work

We have sketched the main features of the first compositional refinement calculus incorporating an assertion language strong enough to express temporal and epistemic notions. While, as we have noted, some further features are required to give a complete treatment of knowledge-based programs in the sense of [4], we already have enough expressiveness in the framework to be able to view knowledge-based programs as special cases of our more general programs using quantified local propositions. Moreover, the derivation we have presented at length is very much in the spirit of the knowledge-based approach. (Indeed, precisely the same implementation is derived in [2].) In contrast to tests for knowledge, tests for local predicates satisfying some extra conditions are more likely, in general, to admit efficient implementations. In future work, we plan to extend the framework of this paper to multiple agents and asynchrony. Ultimately, we hope to achieve a highly expressive, flexible and abstract framework supporting the knowledge-based development of distributed systems.

References

[1] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[2] R. I. Brafman, J.-C. Latombe, Y. Moses, and Y. Shoham. Applications of a logic of knowledge to motion planning under uncertainty. *Journal of the ACM*, 44(5):633–668, Sept. 1997.

[3] K. Engelhardt, R. van der Meyden, and Y. Moses. Knowledge and the logic of local propositions. In I. Gilboa, editor, *Theoretical Aspects of Rationality and Knowledge, Proceedings of the Seventh Conference (TARK 1998)*, pages 29–41. Morgan Kaufmann, July 1998.

[4] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT-Press, 1995.

[5] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.

[6] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, July 1990.

[7] I. Hayes. Separating timing and calculation in real-time refinement. In J. Grundy, M. Schwenke, and T. Vickers, editors, *International Refinement Workshop and Formal Methods Pacific 1998*, Discrete Mathematics and Theoretical Computer Science, pages 1–16. Springer-Verlag, 1998.

[8] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[9] C. C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

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

[11] Y. Moses and O. Kislev. Knowledge-oriented programming. In *Proceeding of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC 93)*, pages 261–270, New York, USA, Aug. 1993. ACM Press.

[12] Y. Moses and M. R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169, 1988.

[13] B. Sanders. A predicate transformer approach to knowledge and knowledge-based protocols. In *Proceeding of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC 91)*, pages 217–230, 19–21 Aug. 1991.

[14] M. Utting and C. Fidge. A real-time refinement calculus that changes only time. In H. Jifeng, J. Cooke, and P. Wallis, editors, *BCS-FACS Seventh Refinement Workshop*. Springer-Verlag, 1996.

[15] R. van der Meyden. Knowledge based programs: On the complexity of perfect recall in finite environments. In Y. Shoham, editor, *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge*, pages 31–50. Morgan Kaufmann, Mar. 17–20 1996.

[16] R. van der Meyden and Y. Moses. On refinement and temporal annotations. `http://www.cse.unsw.edu.au/~meyden/research/temprefine.ps`.

[17] R. van der Meyden and Y. Moses. Top-down considerations on distributed systems. In *Proceedings 12th International Symposium on Distributed Computing, DISC'98*, volume 1499 of *LNCS*, pages 16–19, Sept. 1998. Springer-Verlag.

[18] M. Y. Vardi. Implementing knowledge-basd programs. In Y. Shoham, editor, *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge*, pages 15–30. Morgan Kaufmann, Mar. 17–20 1996.