to appear in Robert Nieuwenhuis and Andrei Voronkov, editors, *8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume ???? of *LNAI*, pages ??–??.
©Springer-Verlag, December 2001.

A Refinement Theory that Supports Reasoning about Knowledge and Time for Synchronous Agents

Kai Engelhardt Ron van der Meyden* Yoram Moses†

October 10, 2001

An expressive semantic framework for program refinement that supports both temporal reasoning and reasoning about the knowledge of multiple agents is developed. The refinement calculus owes the cleanliness of its decomposition rules for all programming language constructs and the relative simplicity of its semantic model to a rigid synchrony assumption which requires all agents and the environment to proceed in lockstep. The new features of the calculus are illustrated in a derivation of the two-phase-commit protocol.

1. Introduction

The knowledge-based approach to the design and analysis of distributed systems, introduced by Halpern and Moses [13] involves the use of modal logics of knowledge. One of the key contributions of this approach is the notion of knowledge-based programs [9, 8], that is, programs with formulas in the logic of knowledge as tests in conditional constructs. Such programs contain statements of the form "if you know that ϕ then do A else B". Knowledge-based programs provide abstractions of distributed programs that allow for perspicuous descriptions of how agents' actions are related to their, typically incomplete, state of information about their environment. While knowledge-based programs provide a useful type of high level specification for distributed systems, they have a number of limitations, such as their failure to abstract actions, the fact that they sometimes permit as implementations only programs of unacceptably high computational complexity, and the lack of a formal development framework [23, 14, 7]. These limitations have led us to seek to develop a more general framework that enhaces knowledge-based programs in order to overcome their limitations. Our more general framework is in the form of a *refinement calculus*. Refinement calculi [1, 19, 20] are formalizations of the ubiquitous stepwise refinement method of program construction [26]. They view programs and specifications as having the same semantic type, and guide top-down development by a set of rules. One begins with a specification and transforms it to an implementation by means of a sequence of correctness preserving refinement steps according to those rules. The initial focus in this area has been on sequential programs and atemporal assertions.

Recently, the scope of refinement calculi has been broadened to overcome both limitations. Some calculi have distributed or parallel systems as domain [2, 3], others admit the expressive power of temporal logics [24, 15].

The refinement calculus we are developing is aimed at supporting development of distributed systems using the insights afforded by the knowledge-based approach, but without suffering the

*School of Computer Science and Engineering, The University of New South Wales, Sydney 2052 Australia, for $X \in \{\mathrm{kaie}, \mathrm{meyden}\}$, email: X@cse.unsw.edu.au, www: http://www.cse.unsw.edu.au/~X/
†Department of Electrical Engineering, Technion, Haifa, Israel, email: moses@ee.technion.ac.il

limitations of knowledge-based programs. Some of the key ingredients of the calculus were introduced in previous papers. In order to address the problem that knowledge-based programs sometimes have only implementations of high computational complexity, [6] generalised multi-agent epistemic logics [13] to the Logic of Local Propositions (LLP), which enables knowledge-based program-like specifications having a broader range of implementations. Since reasoning about distributed systems frequently involves temporal considerations, a refinement calculus incorporating linear time temporal logic assertions was developed in [25]. The logic of local propositions and the temporal refinement calculus were amalgamated in [7], yielding the first refinement calculus that supports assertional reasoning about knowledge and time.

The calculus of [7] is restricted to a single agent interacting with its environment. The present paper presents the next step towards our aim, by extending the single agent calculus to one with multiple agents. We restrict ourselves to a program operator for agent parallelism which amounts to insisting that all agents and even the environment proceed in lockstep. While we will ultimately be interested in more flexible forms of agent interaction, this operator has the benefit of admitting some perspicuous refinement rules, and is already rich enough to capture some interesting examples that exercise key features of our knowledge-based refinement approach. In particular, we demonstrate how it can be used for the top down development of an atomic commitment protocol in the presence of process crashes. This provides an informative case study for our broader program.

The paper is structured as follows. Section 2 summarizes the framework presented in [7] and how it is extended to the synchronous multi-agent case. We first define an assertion language that adapts the LLP semantics to the richer temporal setting required for reasoning about multi-agent programs. Then we present the syntax and semantics of our multi-agent programming and specification language. Section 3 defines the refinement relation we use for this class of programs and develops a number of refinement rules valid for this relation, including the abovementioned decomposition rules. Section 4 illustrates the use of the new rules in a derivation of the two-phase-commit protocol. Section 5 concludes and indicates where we intend to go from here.

2. Programming with Assertions about Knowledge and Time

2.1. Semantic Domain

In [7] we presented a semantic framework that is extended here from the single-agent case to the synchronous multi-agent case. Further motivation for this general line of work can be found in [8].

Let $n \in \mathbb{N}$ be a number of *agents*, traditionally named $1,\ldots,n$. Let L_e be a set of possible states for the environment and, for $i \in \{1,\ldots,n\}$, let L_i be a set of possible local states for agent i. We take $\mathcal{G} = (\times_{i=1}^{n} L_i) \times L_e$ to be the set of *global states*. Let A_i and A_e be nonvoid sets of *actions* for agent i and for the environment, respectively. (These sets usually contain a special *null action* Λ.) A *joint action* is an $(n+1)$-tuple $(a_1,\ldots,a_n,a_e) \in \mathcal{A} = (\times_{i=1}^{n} A_i) \times A_e$. 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 $\text{Points}(S)$ for the set of points of S. Let *Prop* be a set of propositional variables. An *interpretation* of a system S is a mapping $\pi : \mathit{Prop} \longrightarrow 2^{\text{Points}(S)}$ associating a set of points with each propositional variable.[1] Intuitively, proposition $p \in \mathit{Prop}$ is true exactly at the

[1]Some standard treatments such as [8] interpret propositions over sets of global states so that $\pi : \mathit{Prop} \to 2^{\mathcal{G}}$. This could easily be done for this paper, with practically no changes to the resulting framework.

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 an agent's knowledge. We say two points $(r,c),(r',c')$ in a system S are *i-indistinguishable*, denoted $(r,c) \sim_i (r',c')$, if the agent i's local components of the global states at these points are equal, i.e., the ith projection of $h(c)$ and $h'(c')$ are equal, where $r = (h,\alpha)$ and $r' = (h',\alpha')$. A set P of points of S is *i-local* if it is closed under $\sim_i$, in other words, when for all pairs of points $(r,c),(r',c')$ of S, if $(r,c) \in P$ and $(r,c) \sim_i (r',c')$ then $(r',c') \in P$. Intuitively, i-local sets of points correspond to properties that agent i is able to determine entirely on the basis of its local state. If π and π' are interpretations and $p \in \mathit{Prop}$, then π' is said to be a *p-variant* of π, denoted $\pi \simeq_p \pi'$, if π and π' differ at most in the value of p. If, additionally, $\pi'(p)$ is i-local, then π' is said to be an *i-local p-variant* of π, denoted $\pi \simeq^i_p \pi'$. 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 a *p-variant (i-local p-variant)* of $\mathfrak{I}$, denoted $\mathfrak{I} \simeq_p \mathfrak{I}'$ (resp. $\mathfrak{I} \simeq^i_p \mathfrak{I}'$), if $S = S'$ and $\pi \simeq_p \pi'$ (resp. $\pi \simeq^i_p \pi'$).

2.2. An Assertion Language for Knowledge and Time

The assertion language $\mathcal{L}$ we use in this paper resembles S5 with two additions: (a) restricted monadic second order quantification for each of the agent's local predicates and (b) operators from the linear time temporal logic LTL [18]. Its syntax is given by:

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

where $p \in \mathit{Prop}$ and $i \in \{1,\dots,n\}$. 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 $\forall p\,(\phi)$ says that ϕ is true for all assignments of a proposition (set of points) to the propositional variable p. Formula $\forall_i p\,(\phi)$ says the same, except that the assignments to p must be i-local. We write $\exists_i p\,(\phi)$ for its dual $\neg\forall_i p\,(\neg\phi)$. 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*, $\vee$, $\exists p\,(.)$, and definable future time operators such as $\Box$ ("henceforth") and $\Diamond$ ("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 $(r',c') \in \mathrm{Points}(S)$;
- $\mathfrak{I},(r,c) \models \forall p\,(\phi)$ iff $\mathfrak{I}',(r,c) \models \phi$ for all $\mathfrak{I}'$ such that $\mathfrak{I} \simeq_p \mathfrak{I}'$;
- $\mathfrak{I},(r,c) \models \forall_i p\,(\phi)$ iff $\mathfrak{I}',(r,c) \models \phi$ for all $\mathfrak{I}'$ such that $\mathfrak{I} \simeq^i_p \mathfrak{I}'$;
- $\mathfrak{I},(r,c) \models \bigcirc\phi$ 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 $d < e \leq c$.

Let $\phi \in \mathcal{L}$ and let $\mathcal{S}$ be a set of interpreted systems. We say that ϕ *is valid in* $\mathcal{S}$, and write $\mathcal{S} \models \phi$, if ϕ is satisfied at all points of all interpreted systems in $\mathcal{S}$.

It is possible to express in $\mathcal{L}$ many operators from the literature on reasoning about knowledge. For example, consider the standard knowledge operator K_i, defined by $\mathfrak{I},(r,c) \models K_i\phi$ if $\mathfrak{I},(r',c') \models \phi$ for all points (r',c') of $\mathfrak{I}$ such that $(r,c) \sim_i (r',c')$. The formula $K_i\phi$ is expressible as $\exists_i p\,(p \wedge \mathsf{Nec}(p \rightarrow \phi))$, where p does not occur freely in ϕ. This characterization of knowledge motivates a weaker notion, that of an i-local condition p that is sound for ϕ, i.e., satisfies $\mathsf{Nec}(P \rightarrow \phi)$. This weaker notion (which we apply in Section 4), helps to overcome some of the complexity and proof-theoretic limitations of knowledge-based programs. We refer to [6] for further examples and discussion.

2.3. A Programming Language 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.

2.3.1. Syntax.

Let CT be a set of *constraint tags* and let $PV, PV_1, \ldots, PV_n$ be mutually disjoint sets of *program variables*. For each $i \in \{1,\ldots,n\}$ define a syntactic category Prg_i of *(sequential) programs* for agent i by:

$$Prg_i \ni P_i \ ::= \ \varepsilon \mid Z \mid a \mid P_i * P_i \mid P_i + P_i \mid P_i^{\omega} \mid \exists_i p\,(P_i) \mid [\phi,\phi]^X \mid [\phi]^X \mid \{\phi\}_C$$

where $Z \in PV_i$, $a \in A_i$, $p \in Prop$, $\phi \in \mathcal{L}$, $X \in CT$, and $C \subseteq CT$. The very similar syntactic category Prg of *n-agent programs* is formally defined by:

$$Prg \ni P \ ::= \ \varepsilon \mid Z \mid (\|_{i=1}^{n} P_i) \mid P * P \mid P + P \mid P^{\omega} \mid \exists_i p\,(P) \mid [\phi,\phi]^X \mid [\phi]^X \mid \{\phi\}_C$$

where $Z \in PV$, $P_i \in Prg_i$, $p \in Prop$, $\phi \in \mathcal{L}$, $X \in CT$, and $C \subseteq CT$. The intuitive meaning of these constructs is the same for both levels of the language. 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 sequential program for agent i may refer directly to actions a of agent i, 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 program P. The construct $\exists_i p\,(P)$ can also be understood as a kind of nondeterministic choice: it states that P runs with respect to some assignment of an i-local proposition to the propositional variable p. The last three constructs are similar to constructs found in refinement calculi. Intuitively, the *specification* $[\phi,\psi]^X$ stands for any program that, if started at a point satisfying ϕ, eventually terminates at a point satisfying ψ.[2] 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 tag X in specifications and coercions acts as a label that allows references by other pieces of program text. Specifically, this is done in the *assertion statements*, which have the form $\{\phi\}_C$ and act as 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

[2] 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.

fragments, and on specification and coercion statements whose tags are in C. We may omit the constraint tags when there is no need to make such references.

In local programs "$*$" binds tighter than "$+$", which in turn binds tighter than (the necessarily n-agent) "$\|$". Also in n-agent programs "$*$" binds tighter than "$+$" but both bind less tightly than "$\|$". We employ parentheses to indicate aggregation wherever necessary and tend to omit "$*$" near coercions and assertions. We use **if**^{X} ϕ **then** P **else** Q **fi** as abbreviation for $[\phi]^X * P + [\neg\phi]^X * Q$.

2.3.2. Semantics.

Our semantics will treat programs as 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 in Section 3.1.

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

An *execution tree for agent i* is a Prg_i-labelled binary tree, subject to the following constraints on the nodes l:

1. If l is labelled by ε, a program variable $Z \in PV$, a specification $[\phi,\psi]^X$, a coercion $[\phi]^X$, or an assertion $\{\phi\}_C$, then l is a leaf.
2. If l is labelled by a basic action $a \in A_i$, then l is also a leaf.
3. If l is labelled by $\exists_i p\,(P)$ then l has exactly one child $l\cdot 0$, labelled by P.
4. If l is labelled by $P*Q$ or $P+Q$ then l has exactly two children $l\cdot 0$, $l\cdot 1$, labelled by P and Q respectively.
5. If l is labelled by P^ω then l has exactly two children, $l\cdot 0$, $l\cdot 1$, labelled by ε and $P*(P^\omega)$, respectively.

With each program $P_i \in Prg_i$ we associate a particular execution tree, T_{P_i}, namely the unique execution tree for agent i labelled with P_i at the root λ. An *n-agent execution tree* is a $Prg \cup \bigcup_{i=1}^n Prg_i$-labelled n-ary tree, subject to the same constraints on the nodes l as if it were an execution tree for some agent, except for condition 2, which is replaced by:

2' If l is labelled by $(\|_{i=1}^n P_i)$ then l has exactly n children $l\cdot 0,\ldots,l\cdot(n-1)$, labelled by $P_1,\ldots,P_n$ respectively. For each $i \in \{1,\ldots,n\}$ the subtree rooted at $l\cdot(i-1)$ is an execution tree for agent i.

With each n-agent program $P \in Prg$ we associate a particular execution tree, namely the unique n-agent execution tree labelled with P at the root λ.

Interval Sets. 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, iis, by means of certain mappings from execution trees to iis. 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 run-unique 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 $\{\ r[c, f(r[c,d])] \mid r[c,d] \in I\ \}$.

Embeddings. Let S be a system, let (π, I) be an iis w.r.t. S, and let $P_i \in Prg_i$ be a sequential program for agent i. A function θ mapping each node l of T_{P_i} to an iis $(\pi_\theta(l), I_\theta(l))$ is an *embedding* of T_{P_i} in (π, I) w.r.t. S whenever the following conditions are satisfied:

1. $\theta(\lambda) = (\pi, I)$.
2. If l is labelled ε or $\{\phi\}_C$, then $c = d$ for all $r[c,d] \in I_\theta(l)$.
3. If l is labelled a for some $a \in A_i$, then, for all $(h,\alpha)[c,d] \in I_\theta(l)$, if $c < \infty$ then both $d = 1 + c$ and the ith projection of $\alpha(c)$ is a.
4. If l is labelled $[\phi,\psi]^X$, then, for all $r[c,d] \in I_\theta(l)$, if $c < \infty$ and $(S, \pi_\theta(l)), (r,c) \models \phi$, then both $d < \infty$ and $(S, \pi_\theta(l)), (r,d) \models \psi$.
5. If l is labelled $[\phi]^X$, then $c < \infty$ implies that $c = d$ and $(S, \pi_\theta(l)), (r,c) \models \phi$, for all $r[c,d] \in I_\theta(l)$.
6. If l is labelled $\exists_i p\,(Q)$ then $\pi_\theta(l) \simeq^i_p \pi_\theta(l \cdot 0)$ and $I_\theta(l \cdot 0) = I_\theta(l)$.
7. If l is labelled $Q_1 + Q_2$, then $\pi_\theta(l \cdot 0) = \pi_\theta(l \cdot 1) = \pi_\theta(l)$ and $I_\theta(l)$ is the disjoint union of $I_\theta(l \cdot 0)$ and $I_\theta(l \cdot 1)$.
8. If l is labelled $Q_1 * Q_2$, then $\pi_\theta(l \cdot 0) = \pi_\theta(l \cdot 1) = \pi_\theta(l)$ and there is an f dividing $I_\theta(l)$ such that $I_\theta(l \cdot 0) = f_\blacktriangleright(I_\theta(l))$ and $I_\theta(l \cdot 1) = f_\blacktriangleleft(I_\theta(l))$.
9. If l is labelled Q^ω then $\pi_\theta(l \cdot 0) = \pi_\theta(l \cdot 1) = \pi_\theta(l)$ and $I_\theta(l)$ is the disjoint union of $I_\theta(l \cdot 0)$ and $I_\theta(l \cdot 1)$ (as in case 7) and, for all $r[c,d] \in I_\theta(l)$:
   $d = \bigsqcup\{\ d' \mid r[c',d'] \in I_\theta(l \cdot m)\ \text{ for some leaf } l \cdot m \text{ of } T_P \text{ below } l\ \}$.

We write $S, (\pi, I) \Vdash_\theta P_i$ whenever θ is an embedding of T_{P_i} in (π, I) w.r.t. S. Say that P_i *occurs over* (π, I) *w.r.t.* S and write $S, (\pi, I) \Vdash P_i$ if there exists a θ such that $S, (\pi, I) \Vdash_\theta P_i$. (See [7] for further motivation on the definitions for single agent related notions.)

Let $P \in Prg$. A function θ mapping each node l of T_P to an iis $(\pi_\theta(l), I_\theta(l))$, respectively, is an *embedding* of T_P in (π, I) w.r.t. S whenever the above conditions with the following replacement for condition 3 are satisfied:

3' If l is labelled by $(\|_{i=1}^n P_i)$ then $\theta(l \cdot (i-1)) = \theta(l)$ and $S, \theta(l \cdot (i-1)) \Vdash P_i$ for all $i \in \{1, \ldots, n\}$.

Analogously to the single-agent case, 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 and write $S, (\pi, I) \Vdash P$ if there exists a θ such that $S, (\pi, I) \Vdash_\theta P$.

3. Refinement

The semantics presented above is a straightforward generalization of the semantics of [7] which corresponds to the special case $n = 1$. That paper also contains an example illustrating the need for sets of intervals instead of single intervals in the definition of program semantics. In the context of knowledge-based programs, the need for sets (in that case of runs, not intervals) has been observed by Moses in his thesis [21] and is discussed by Halpern [12].

Traditionally, a program P refines another program Q whenever the use of P does not lead to an observation which is not also an observation of Q [10]. A refinement relation of this type, when it is 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. The properties of the refinement relations then guarantee that the final stage of the derivation refines the initial stage.

The standard approach to define a refinement relation based on our semantics is to identify the notion of observation with occurrence, that is to say that *P refines Q*, denoted $P \sqsubseteq Q$, when for all systems S and iis (π, I) over S, if $S, (\pi, I) \Vdash P$ then $S, (\pi, I) \Vdash Q$. While having both desired properties, this refinement notion is insensitive to assertion statements because the semantics cannot distinguish between programs that differ only in their assertions. Assertions are meant to play an important role in our refinement rules: they should allow us to specify assumptions about the context a program fragment is used in.

3.1. The role of assertions

We now briefly summarize the role of assertions $\{\phi\}_C$ in the framework and define the associated semantic notions. The reader is referred to [25] for a more detailed explanation of these ideas in a simpler setting and to [7] for additional refinement rules.

Constraint erasure. In order to capture constraint dependencies, we firstly define for each program P and constraint tag set $C \subseteq CT$ a program $P^{[C]}$ that is the same as P, except that only constraints whose tags are in C are enforced: all other constraints are relaxed. We formally obtain $P^{[C]}$ 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 $X \notin C$ in P by $[\mathit{false}, \mathit{true}]^X$ in $P^{[C]}$.

Program validity. Secondly, we may now define a program (regardless of whether it is a sequential program or an n-agent program) P to be *valid* with respect to a set of interpreted systems $\mathcal{S}$ (and write $\models_{\mathcal{S}} P$ for this) when for all assertions $\{\phi\}_C$ in P, all interpreted systems $(S, \pi) \in \mathcal{S}$ and all run-unique interval sets I over S, all embeddings θ of $T_{P^{[C]}}$ into $S, (I, \pi)$ have the property that for all $i \in \{1, \ldots, n\}$ and nodes l of $T_{P_i^{[C]}}$ labelled with $\{\phi\}_C$, we have $S, \theta_i(l) \Vdash [\phi]$. 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 Section 4.

Justification transformations. To define a refinement notion that only allows one to derive valid programs from valid programs, we first need to define a technical notion. A *justification transformation* is a mapping $\eta : CT \longrightarrow 2^{CT}$ that satisfies $X \in \eta(X)$ for all $X \in CT$. In some definitions in the sequel we make liberal use of the pointwise extension of justification transformations: $\eta(C) = \bigcup_{X \in C} \eta(X)$. For instance, 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 $\{\phi\}_{\eta(C)}$.

The *composition* of two justification transformations, η and η', is defined by $\eta \circ \eta' = \lambda X : CT.\eta'(\eta(X))$.

The identity justification transformation $\lambda X : CT.\{X\}$ is denoted by ι. We will also represent justification transformations using expressions of the form $X \hookrightarrow D$, where $X \in CT$ and $D \subseteq CT$. Such an expression denotes the justification transformation that is everywhere as ι, except for on X, which is mapped to $\{X\} \cup D$.

Valid refinement. Let $\mathcal{S}$ be a set of interpreted systems, let η be a justification transformation and let P and Q be sequential programs for agent i. We say that *P validly refines Q in $\mathcal{S}$ under* η, and write $P \leq^{\mathcal{S}}_{\eta} Q$, if for all programs $R \in Prg$ and program variables $Z \in PV_i$, if $R[^Q/_Z]$ is valid with respect to $\mathcal{S}$ then $R\eta[^P/_Z]$ is valid with respect to $\mathcal{S}$, and for all $(S,\pi) \in \mathcal{S}$ and run-unique interval sets I over S, if $S,(\pi,I) \Vdash R\eta[^P/_Z]$ then $S,(\pi,I) \Vdash R[^Q/_Z]$.

An analogous definition fixes the notion of an n-agent program P validly refining $Q \in Prg$. The only other change in the previous paragraph is that program variable Z must now be drawn from PV.

3.2. 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 n-agent programs, and refer to [7] for additional rules concerning the sequential programs, which are also sound in the framework of the present paper. Formally, most rules should come in two versions: one for sequential programs and one for n-agent programs, but in many cases the two will be identical but for typing of the components, so we generally omit the distinction.

Two rules are essential ingredients of any refinement calculus. Valid refinement must be transitive and the program constructs must be monotone w.r.t. the valid refinement order. These two rules make it possible for refinement to be broken down into a sequence of steps that operate on small program fragments.

$$\frac{P \leq^{\mathcal{S}}_{\eta} Q, \quad Q \leq^{\mathcal{S}}_{\eta'} R}{P \leq^{\mathcal{S}}_{\eta\circ\eta'} R} \qquad \textbf{trans}$$

$$\frac{P \leq^{\mathcal{S}}_{\eta} Q}{R\eta[^P/_Z] \leq^{\mathcal{S}}_{\eta} R[^Q/_Z]} \qquad \textbf{mon}$$

One advantage of our model is that specifications can be split using any of the composition operators $+$, $*$, and $\|$.

$$\frac{\mathcal{S} \models \psi_1 \vee \psi_2 \to \psi}{[\phi,\psi_1]^{X_1} + [\phi,\psi_2]^{X_2} \leq^{\mathcal{S}}_{X\hookrightarrow\{X_1,X_2\}} [\phi,\psi]^X} \qquad \textbf{+-spec}$$

$$\frac{\mathcal{S} \models \phi \to \phi_1 \vee \phi_2, \quad \mathcal{S} \models \psi_1 \vee \psi_2 \to \psi}{\textbf{if}^{X_1}\ \phi_1\ \textbf{then}\ [\phi_1,\psi_1]^{X_2}\ \textbf{else}\ [\phi_2,\psi_2]^{X_3}\ \textbf{fi} \leq^{\mathcal{S}}_{X\hookrightarrow\{X_1,X_2,X_3\}} [\phi,\psi]^X} \qquad \textbf{if-spec}$$

$$[\phi,\phi']^{X_1} * [\phi',\psi]^{X_2} \leq^{\mathcal{S}}_{X\hookrightarrow\{X_1,X_2\}} [\phi,\psi]^X \qquad \textbf{*-spec}$$

$$\frac{\mathcal{S} \models \phi \to \bigwedge_{i=1}^n \phi_i, \quad \mathcal{S} \models \bigwedge_{i=1}^n \psi_i \to \psi}{(\|_{i=1}^n [\phi_i,\psi_i]^{X_i}) \leq^{\mathcal{S}}_{X\hookrightarrow\{X_1,\ldots,X_n\}} [\phi,\psi]^X} \qquad \textbf{\|-spec}$$

The execution of many distributed programs can be meaningfully split into several *phases*. Specifying, or reasoning about, such programs is potentially simplified if one can do so on a phase-by-phase basis. Instead of reasoning about the parallel composition of local programs each of which comprises of a sequence of phases, it is usually much easier to reason about the sequential composition of several global phases, each of which is an n-agent program. Such phased reasoning would be supported by a rule of the form

$$(\|_{i=1}^{n}(P_i * Q_i)) \leq_{\iota}^{\mathcal{S}} (\|_{i=1}^{n} P_i) * (\|_{i=1}^{n} Q_i)$$

Refinement rules reflecting this proof principle do not hold without side conditions on $(\|_{i=1}^{n} P_i)$, $(\|_{i=1}^{n} Q_i)$, and $\mathcal{S}$ (though the converse holds unconditionally.) One such condition is that all the intervals in the iis over $\mathcal{S}$ over which some P_i occurs have the same length.[3] Say that a program P *has length* l *in* $\mathcal{S}$ whenever, for each $(S,\pi) \in \mathcal{S}$, for each run-unique set I of intervals over S, if $S,(\pi,I) \Vdash P$, interval $r[c,d] \in I$, and c is finite, then $l = d - c$. We say that P *has constant length in* $\mathcal{S}$ if there exists $l \in \mathbb{N}_+$ such that P has length l in $\mathcal{S}$.

$$\frac{\text{all } P_i \text{ have the same length in } \mathcal{S}}{(\|_{i=1}^{n}(P_i * Q_i)) \leq_{\iota}^{\mathcal{S}} (\|_{i=1}^{n} P_i) * (\|_{i=1}^{n} Q_i)} \qquad \textbf{$\|$-distribute-$*$}$$

Having constant length is a semantic condition. A sufficient (but not necessary) syntactic condition is that the program does not contain any specification statement, iteration, or choice. This can be relaxed in various ways. For instance, choice $P+Q$ itself is not problematic if both P and Q have the same length in $\mathcal{S}$. Similarly, iteration P^{ω} has constant length if P does and the number of iteration is fixed, e.g., in the style of a **for**-loop. Even specification statements can be confined to constant length with the aid of existential quantification over propositional variables. In [7] we described *single-stepping* specifications, which are programs of length 1. In general, one can refine a specification statement by adding a constraint that the specified pre/post behaviour is achieved in exactly l steps:

$$\exists p, p' \left(\left[\mathsf{only}\, p \wedge \bigcirc^{l} \mathsf{only}\, p' \right]^{X_1} [true, p' \wedge (\diamondminus (p \wedge \phi) \to \psi)]^{X} \right) \leq_{X \hookrightarrow \{X_1\}}^{\mathcal{S}} [\phi, \psi]^{X} \qquad \textbf{spec-length}$$

where $\bigcirc^{l}$ stands for l operators $\bigcirc$ and $\mathsf{only}\,\phi$ is an abbreviation for $\neg \circleddash \diamondminus \phi \wedge \phi \wedge \neg \bigcirc \Diamond \phi$ which holds exactly at the unique point in a run that makes ϕ true, if there is such a point.

4. Example: Two-Phase Commit

To illustrate our framework we consider a problem from the area of distributed databases. Two- and three-phased commit protocols have previously been considered (though less formally) from a knowledge-based perspective in [11, 22, 16].

Problem. Suppose n agents collaborate on a database transaction. After each of the agents has done its share of the transaction, they want to come to an agreement on whether to commit the transaction for later use of the result by other transactions. Each agent can be assumed to have formed an initial vote but not yet made a final decision. All that remains to be done is to ensure that no two agents make different decisions. The only faults considered in this example are crashes — communication is assumed reliable. The presentation is inspired by [4, 17].

[3] Another such side condition in the literature is that each phase is *communication-closed* [5]. This means that there is no communication across phase boundaries. The communication-closed layers rule is tailored to an asynchronous setting, rather than our synchronous framework.

Informal specification. The requirements on an atomic commitment protocol are typically stated as follows:

1. All agents that reach a decision reach the same one.
2. If an agent decides to commit, then all agents voted to commit.
3. If there are no failures and all agents voted to commit, then the decision reached is to commit.

Algorithm. The standard solution to this problem is called *two-phase commit (2PC)*. One distinguished agent, say, agent 1, collects the other agents' votes. If all votes (including agent 1's) are "commit" then agent 1 tells all other agents to commit, otherwise, to abort. Note that "otherwise" could also mean that some agent crashed before being able to communicate its vote to agent 1.

The first phase thus consists of all agents telling agent 1 their individual votes and, if they are in favour of aborting, deciding to do so. In the second phase the agents learn (if they haven't done so already because their vote was "abort") what to decide.

Assumptions about $\mathcal{S}$. Agent i's vote is modeled as the value of a local Boolean variable, v_i, which is either *true* (for "commit") or *false* (for "abort"). Agent i's decision d_i has one more option, it could also be $\perp$ (for "undecided"). Agents never change their initially held votes. Initially, agents are undecided. Once they decide, they cannot change their decision. Each agent $i = 2, \ldots, n$ has a three-valued variable c_i in which it can receive a message from agent 1. Agent 1 also has a set of three-valued variables $m_2, \ldots, m_n$ in which it receives information from the other agents concerning their vote.[4] Initially, each m_i has value $\perp$.

Agents have available the following actions. The environment is used to model the fact that all these actions have a non-deterministic effect on the state. When an action is performed, it either has its intended effect on the state, or (at the discretion of the environment) the agent crashes, and the action has no effect on the state. Once an agent has crashed, it is crashed at all subsequent states, so its actions have no effect. We describe just the intended effects below. All agents can perform the null action Λ, which has no effect. Each agent i also has the action $decide_i(v)$, where v is a truth value, the intended effect of which is to set d_i equal to v. In addition to these actions, agent 1 can perform the action $broadcast_1(v)$, where v is a truth value. The action $broadcast_1(v)$ has the intended effect of setting all the variables c_i equal to the value of v. Each agent i can perform the action $send_{i,1}(v)$, where v is a truth value. The intended effect of this is to set agent 1's variable m_i equal to v.

The environment has a set of actions $crash(F)$, where F is a subset of $\{1, \ldots, n\}$, which it uses to crash the agents in F, making the proposition $\dagger_i$ true for agents $i \in F$. Initially, $\dagger_i$ is false. Crashes are non-recoverable: once it has perfomed $crash(F)$, the environment always performs $crash(F')$ for some superset F' of F, so that once crashed, an agent $i \in F$ remains crashed (and the proposition $\dagger_i$ remains true.)

Due to space restrictions, we restrict ourselves to this rather informal description of $\mathcal{S}$. One of the strengths of the full calculus is that each formal description of the set of target systems generates a set of tailor-made refinement rules. (See [7] for some detail.)

4.1. Sketch of a 2PC protocol derivation

The point of departure of this derivation is an assertion followed by a single specification. The assertion expresses that this program should only be used in contexts that guarantee that the pro-

[4] One could of course, use a richer modelling of communication, such as message buffers, but this simple approach suffices for our illustrative purposes.

gram is started in an initial state. The specification statement expresses that the program should take the agents from an initial state to a state satisfying the informal specification above. The precondition of the initial specification expresses that no agent has received a message and that no agent has decided. Let us abbreviate: *none voted* $\mathrm{NV} = \bigwedge_{i=2}^{n}(m_i = \bot)$, *none received broadcast* $\mathrm{NB} = \bigwedge_{i=2}^{n}(c_i = \bot)$, and *none decided* $\mathrm{ND} = \bigwedge_{i=1}^{n}(d_i = \bot)$.

$$\mathrm{Init} = \mathrm{NV} \wedge \mathrm{NB} \wedge \mathrm{ND} \ .$$

Note that we do not assume that all agents are alive. The postcondition of the initial specification is $\exists d\,(\Psi)$ where

$$\Psi = \textstyle\bigwedge_{i=1}^{n}(d_i \neq \neg d) \wedge (d \rightarrow \mathrm{AC}) \wedge (\mathrm{AC} \wedge \mathrm{NF} \rightarrow \bigwedge_{i=1}^{n} d_i) \ .$$

In the postcondition, each of the three conjuncts corresponds to one of the informal requirements stated above. One of the first refinement steps will split the specification into two phases. The challenge there is to find a good intermediate assertion. To motivate our choice of intermediate assertion, we note the following informal argument. Intuitively, we would like the first phase of the protocol to consist of each agent informing the coordinator of its vote, so that by the end of the first phase, assuming there have been no failures, if all agents vote to commit then by the end of the first phase the coordinator knows this. That is, we would like to make true the formula $\mathrm{NF} \wedge \mathrm{AC} \rightarrow K_1\mathrm{AC}$, in which we employ the abbreviations *none failed* $\mathrm{NF} = \bigwedge_{i=1}^{n} \neg \dagger_i$ and *all commited* $\mathrm{AC} = \bigwedge_{i=1}^{n} v_i$ true. Recall from the discussion above that $K_1\mathrm{AC}$ is equivalent to $\exists_1 p\,(p \wedge \mathsf{Nec}(p \rightarrow \mathrm{AC}))$, i.e., the agent knows that AC if some sound 1-local test p for AC is true. This motivates rewriting the intermediate assertion as the following slightly stronger formula

$$\Phi = (\mathrm{NF} \wedge \mathrm{AC} \rightarrow p) \wedge (p \rightarrow \mathrm{AC}) \wedge \mathrm{ND}$$

where we have dropped the necessity operator, because the formula as a whole will be implicitly given a neccessity-like force by placing it in a specification construct. To capture that the proposition p in the intermediate assertion is required to be 1-local, we add a quantifier as the first refinement step.

$$\begin{array}{ll}
 & \{\mathrm{Init}\}_{\{Z\}} * [\mathrm{Init}, \exists d\,(\Psi)]^X \\
\geq_{\iota}^{\mathcal{S}} & \quad \text{(instance of axiom: } \exists_i p\,(P) \leq_{\iota}^{\mathcal{S}} P \text{, for } p \text{ not free in } P \text{)} \\
 & \exists_1 p \left(\{\mathrm{Init}\}_{\{Z\}} * [\mathrm{Init}, \exists d\,(\Psi)]^X \right) \\
\geq_{X \hookrightarrow \{Y\}}^{\mathcal{S}} & \quad (*\textbf{-spec}) \\
 & \exists_1 p \left(\{\mathrm{Init}\}_{\{Z\}} * [\mathrm{Init}, \Phi]^X * [\Phi, \exists d\,(\Psi)]^Y \right)
\end{array}$$

For the moment, let us treat the two phases separately, beginning with the first. After this first phase the obvious candidate expression for p is the *accumulated vote* $\mathrm{AV} = (v_1 \wedge \bigwedge_{i=2}^{n}(m_i = \mathit{true}))$.

$$\begin{array}{ll}
 & \{\mathrm{Init}\}_{\{Z\}} * [\mathrm{Init}, \Phi]^X \\
\geq_{X \hookrightarrow \{X'\}} & \quad \text{(instance of axiom: } [\phi, \psi']^Y\, [\psi]^X \leq_{Y \hookrightarrow \{X\}}^{\mathcal{S}} [\phi, \psi]^Y \text{)} \\
 & \{\mathrm{Init}\}_{\{Z\}} * [\mathrm{Init}, \Phi[^{\mathrm{AV}}/_p]]^X * [\Phi]^{X'} \\
\geq_{\iota}^{\mathcal{S}} & \quad \text{(specification consequence)} \\
 & [\mathrm{Init}, \Phi[^{\mathrm{AV}}/_p]]^X * \left\{\Phi[^{\mathrm{AV}}/_p]\right\}_{\{Z,X\}} * [\Phi]^{X'}
\end{array}$$

To apply the parallel decomposition rule **$\|$-spec** one has to show that in $\mathcal{S}$ the conjunction of the local postconditions implies the original n-agent postcondition $\Phi[{}^{\mathrm{AV}}/_{p}]$, which, in this instance, is straightforward. To abbreviate the connection between crashes, messages, and votes, let us employ the abbreviation $\mathrm{CMV}_i = ((\dagger_i \to (m_i = \bot)) \wedge (\neg\dagger_i \to (m_i = v_i)))$.

$$\begin{aligned}
\geq^{\mathcal{S}}_{\iota} \quad & (\|\textbf{-spec}) \\
& ([\mathrm{Init}, d_1 = \bot]^X \parallel \|_{i=2}^{n} [\mathrm{Init}, \mathrm{CMV}_i \wedge (d_i = \bot)]^X) * \{\Phi[{}^{\mathrm{AV}}/_{p}]\}_{\{Z,X\}} * [\Phi]^{X'} \\
\geq^{\mathcal{S}}_{\iota} \quad & (\text{exploit } \mathcal{S} \text{ to implement local specifications with basic actions}) \\
& (\Lambda \parallel \|_{i=2}^{n} \mathit{send}_{i,1}(v_i)) * \{\Phi[{}^{\mathrm{AV}}/_{p}]\}_{\{Z,X\}} * [\Phi]^{X'}
\end{aligned}$$

Let us now consider the development of the second phase of the program. The intuition here is that the coordinator broadcasts its knowledge of the outcome of the vote and then decides. Each remaining agent i has two options: either it already voted to abort, and thus knows that the only possible decision is to abort, or it voted to commit and has to wait for the broadcast message from agent 1 before it decides according to that message. Consequently the next refinement step is another sequential decomposition. The intermediate assertion $\Phi' = \bigwedge_{i=1}^{n} \phi'_i$ states that, unless agent 1 failed, each uncrashed agent received the broadcast prospective decision:

$$\phi'_1 = (d_1 = \bot) \wedge \textstyle\bigwedge_{i=2}^{n}((\dagger_1 \vee \dagger_i \to c_i = \bot) \wedge (\neg(\dagger_1 \vee \dagger_i) \to c_i = p))$$

It also states that each uncrashed agent $i = 2, \ldots, n$ that voted to abort decided so already:

$$\phi'_i = (\dagger_i \vee v_i \to (d_i = \bot)) \wedge (\neg(\dagger_i \vee v_i) \to \neg d_i)$$

The derivation proper follows.

$$\begin{aligned}
& [\Phi]^{X'} * [\Phi, \exists d\,(\Psi)]^Y \\
\geq^{\mathcal{S}}_{Y \hookrightarrow \{Y'\}} \quad & (*\textbf{-spec}) \\
& [\Phi]^{X'} * [\Phi, \Phi']^Y * [\Phi', \exists d\,(\Psi)]^{Y'} \\
\geq^{\mathcal{S}}_{\iota} \quad & (\text{introduce assertion}) \\
& [\Phi]^{X'} * \{\Phi\}_{\{X'\}} * [\Phi, \Phi']^Y * [\Phi', \exists d\,(\Psi)]^{Y'} \\
\geq^{\mathcal{S}}_{\iota} \quad & (\text{specification consequence}) \\
& [\Phi]^{X'} * [\Phi, \Phi']^Y * \{\Phi'\}_{\{X',Y\}} * [\Phi', \exists d\,(\Psi)]^{Y'}
\end{aligned}$$

Let us focus on the first half of the second phase and proceed by parallel decomposition.

$$\begin{aligned}
& [\Phi]^{X'} * [\Phi, \Phi']^Y \\
\geq^{\mathcal{S}}_{\iota} \quad & (\|\textbf{-spec}) \\
& [\Phi]^{X'} * \|_{i=1}^{n} [\Phi, \phi'_i]^Y \\
\geq^{\mathcal{S}}_{\iota} \quad & (n-1 \text{ times } \textbf{if-spec}) \\
& [\Phi]^{X'} * ([\Phi, \phi'_1]^Y \parallel \|_{i=2}^{n} \textbf{if}^Y\ \neg v_i\ \textbf{then}\ [\neg v_i \wedge \Phi, \phi'_i]^Y\ \textbf{else}\ [v_i \wedge \Phi, \phi'_i]^Y\ \textbf{fi}) \\
\geq^{\mathcal{S}}_{\iota} \quad & (\text{exploit } \mathcal{S} \text{ to implement specifications by basic actions}) \\
& [\Phi]^{X'} * (\mathit{broadcast}_1(p) \parallel \|_{i=2}^{n} \textbf{if}^Y\ \neg v_i\ \textbf{then}\ \mathit{decide}_i(\mathit{false})\ \textbf{else}\ \Lambda\ \textbf{fi})
\end{aligned}$$

In the second half of the second phase all uncrashed yet undecided agents decide.

$$
\begin{array}{ll}
& [\Phi', \exists d\,(\Psi)]^{Y'} \\
\geq_\iota^{\mathcal{S}} & \quad \text{(strengthen postcondition)} \\
& [\Phi', \Psi[^p/_d]]^{Y'} \\
\geq_\iota^{\mathcal{S}} & \quad (\|\textbf{-spec}) \\
& \|_{i=1}^{n} [\Phi', (d_i \neq \neg p) \wedge (p \to \mathrm{AC}) \wedge (\mathrm{AC} \wedge \mathrm{NF} \to d_i)]^{Y'} \\
\geq_\iota^{\mathcal{S}} & \quad (n-1 \text{ times } \textbf{if-spec}) \\
& [\Phi', (d_1 \neq \neg p) \wedge (p \to \mathrm{AC}) \wedge (\mathrm{AC} \wedge \mathrm{NF} \to d_1)]^{Y'} \; \| \\
& \|_{i=2}^{n} \,\textbf{if}^{Y'} \neg d_i \;\textbf{then}\; [\neg d_i \wedge \Phi', (d_i \neq \neg p) \wedge (\mathrm{AC} \wedge \mathrm{NF} \to d_i)]^{Y'} \\
& \qquad\qquad \textbf{else}\; [(d_i \neq \mathit{false}) \wedge \Phi', (d_i \neq \neg p) \wedge (\mathrm{AC} \wedge \mathrm{NF} \to d_i)]^{Y'} \;\textbf{fi} \\
\geq_\iota^{\mathcal{S}} & \quad \text{(exploit } \mathcal{S} \text{ to implement a specification by a basic action)} \\
& \mathit{decide}_1(p) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y'} \neg d_i \;\textbf{then}\; \Lambda \;\textbf{else}\; \mathit{decide}_i(c_i) \;\textbf{fi}
\end{array}
$$

Finally, we have to tie the phases together and get rid of the local propositional quantifier.

$$
\begin{array}{ll}
& \{\mathrm{Init}\}_{\{Z\}} * [\mathrm{Init}, \exists d\,(\Psi)]^X \\
\geq_\iota^{\mathcal{S}} & \quad \text{(by the above and } \textbf{mon}) \\
& \exists_1 p \left(\begin{array}{l}
(\Lambda \;\|\; \|_{i=2}^{n} \mathit{send}_{i,1}(v_i)) * \left\{\Phi[^{\mathrm{AV}}/_p]\right\}_{\{Z,X\}} * [\Phi]^{X'} * \\
(\mathit{broadcast}_1(p) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y} \neg v_i \;\textbf{then}\; \mathit{decide}_i(\mathit{false}) \;\textbf{else}\; \Lambda \;\textbf{fi}) * \\
(\mathit{decide}_1(p) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y'} \neg d_i \;\textbf{then}\; \Lambda \;\textbf{else}\; \mathit{decide}_i(c_i) \;\textbf{fi})
\end{array} \right)
\end{array}
$$

This leaves us with a program that is concrete except for the coercion defining p. This can the be eliminated by pushing the existential quantifier first around this coercion, then inside it, and noting that what results is a tautology, so the coercion can be eliminated.

$$
\begin{array}{ll}
\geq_\iota^{\mathcal{S}} & \quad (p \text{ not free in parts of the program}) \\
& (\Lambda \;\|\; \|_{i=2}^{n} \mathit{send}_{i,1}(v_i)) * \left\{\Phi[^{\mathrm{AV}}/_p]\right\}_{\{Z,X\}} * \\
& \exists_1 p \left(\begin{array}{l}
[\Phi]^{X'} * (\mathit{broadcast}_1(p) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y} \neg v_i \;\textbf{then}\; \mathit{decide}_i(\mathit{false}) \;\textbf{else}\; \Lambda \;\textbf{fi}) * \\
(\mathit{decide}_1(p) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y'} \neg d_i \;\textbf{then}\; \Lambda \;\textbf{else}\; \mathit{decide}_i(c_i) \;\textbf{fi})
\end{array} \right) \\
\geq_\iota^{\mathcal{S}} & \quad \text{(instance of axiom: } [\exists_i p\,(\mathsf{Nec}(\phi \equiv p))]^X \,(P[^\phi/_p]) \leq_\iota^{\mathcal{S}} \exists_i p\,(P)) \\
& (\Lambda \;\|\; \|_{i=2}^{n} \mathit{send}_{i,1}(v_i)) * \left\{\Phi[^{\mathrm{AV}}/_p]\right\}_{\{Z,X\}} * [\exists_1 p\,(\mathsf{Nec}(p \equiv \mathrm{AV}))]^V * \left[\Phi[^{\mathrm{AV}}/_p]\right]^{X'} * \\
& (\mathit{broadcast}_1(\mathrm{AV}) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y} \neg v_i \;\textbf{then}\; \mathit{decide}_i(\mathit{false}) \;\textbf{else}\; \Lambda \;\textbf{fi}) * \\
& (\mathit{decide}_1(\mathrm{AV}) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y'} \neg d_i \;\textbf{then}\; \Lambda \;\textbf{else}\; \mathit{decide}_i(c_i) \;\textbf{fi}) \\
\geq_\iota^{\mathcal{S}} & \quad \text{(all variables in AV are 1-local in } \mathcal{S}\text{, eliminate coercion)} \\
& (\Lambda \;\|\; \|_{i=2}^{n} \mathit{send}_{i,1}(v_i)) * \left\{\Phi[^{\mathrm{AV}}/_p]\right\}_{\{Z,X\}} * \left[\Phi[^{\mathrm{AV}}/_p]\right]^{X'} * \\
& (\mathit{broadcast}_1(\mathrm{AV}) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y} \neg v_i \;\textbf{then}\; \mathit{decide}_i(\mathit{false}) \;\textbf{else}\; \Lambda \;\textbf{fi}) * \\
& (\mathit{decide}_1(\mathrm{AV}) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y'} \neg d_i \;\textbf{then}\; \Lambda \;\textbf{else}\; \mathit{decide}_i(c_i) \;\textbf{fi}) \\
\geq_{X' \hookrightarrow \{Z,X'\}} & \quad \text{(eliminate coercion)} \\
& (\Lambda \;\|\; \|_{i=2}^{n} \mathit{send}_{i,1}(v_i)) * \\
& (\mathit{broadcast}_1(\mathrm{AV}) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y} \neg v_i \;\textbf{then}\; \mathit{decide}_i(\mathit{false}) \;\textbf{else}\; \Lambda \;\textbf{fi}) * \\
& (\mathit{decide}_1(\mathrm{AV}) \;\|\; \|_{i=2}^{n} \,\textbf{if}^{Y'} \neg d_i \;\textbf{then}\; \Lambda \;\textbf{else}\; \mathit{decide}_i(c_i) \;\textbf{fi})
\end{array}
$$

We have now arrived at a concrete program implementing the specification for agent i. While sketchy and a little tedious, the derivation does serve to highlight a number of key features of the framework: the role of reasoning using assertions and coercions, and the use of i-locality assumptions to arrive at conclusions about agents' knowledge.

$$\begin{array}{ll}
\geq_\iota^{\mathcal{S}} & \text{(twice } \|\textbf{-distribute-}{*} \text{ using that all basic actions have length 1)} \\
& (\Lambda * \mathit{broadcast}_1(\mathrm{AV}) * \mathit{decide}_1(\mathrm{AV})) \parallel \\
& \|_{i=2}^{n} \begin{pmatrix} \mathit{send}_{i,1}(v_i) * \textbf{if}^Y \neg v_i \textbf{ then } \mathit{decide}_i(\mathit{false}) \textbf{ else } \Lambda \textbf{ fi} * \\ \textbf{if}^{Y'} \neg d_i \textbf{ then } \Lambda \textbf{ else } \mathit{decide}_i(c_i) \textbf{ fi} \end{pmatrix} \\
\geq_\iota^{\mathcal{S}} & \text{(exploit } \mathcal{S} \text{ to merge the two conditionals)} \\
& (\Lambda * \mathit{broadcast}_1(\mathrm{AV}) * \mathit{decide}_1(\mathrm{AV})) \parallel \\
& \|_{i=2}^{n}(\mathit{send}_{i,1}(v_i) * \textbf{if}^Y \neg v_i \textbf{ then } \mathit{decide}_i(\mathit{false}) * \Lambda \textbf{ else } \Lambda * \mathit{decide}_i(c_i) \textbf{ fi})
\end{array} \quad \blacksquare$$

Observe that the program obtained is only one of many possible implementations of the specification w.r.t. $\mathcal{S}$. Another — in a sense the weakest — program satisfying the specification is one in which there is no attempt made to have agents abort when they know that this is what the decision is going to be but another agent has failed. At the other end of the spectrum, one could implement the specification by 3-phase commit, that is, an extension of 2PC that ensures that no uncrashed agent will block.

5. Conclusion and Future Work

We have sketched the main features of an extension of our compositional refinement calculus to multiple agents. It incorporates an assertion language strong enough to express temporal and epistemic notions. The example sketched in Section 4 was chosen to illustrate the use of the new rules, not those carried over from the single-agent calculus.

In future work, we plan to explore variants of the framework with less rigidly synchronous and fully asynchronous semantics.

Acknowledgment

Manuel Chakravarty provided valuable comments on a draft of this paper.

References

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

[2] R. R. J. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13:133–180, 1990.

[3] R. R. J. Back and Q. Xu. Refinement of fair action systems. *Acta Informatica*, 35(11):131–165, 1998.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155–173, Dec. 1982.

[6] 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.

[7] K. Engelhardt, R. van der Meyden, and Y. Moses. A program refinement framework supporting reasoning about knowledge and time. In J. Tiuryn, editor, *Foundations of Software Science and Computation Structures*, volume 1784 of *LNCS*, pages 114–129. Springer-Verlag, Mar. 2000.

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

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

[10] P. H. B. Gardiner and C. C. Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5(4):367–382, 1993.

[11] V. Hadzilacos. A knowledge-theoretic analysis of atomic commitment protocols. In *Proceedings 6th ACM Symposium on Principles of Database Systems*, pages 129–134, 1987.

[12] J. Y. Halpern. A note on knowledge-based programs and specifications. *Distributed Computing*, 13(3):145–153, 2000.

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

[14] J. Y. Halpern and Y. Moses. Using counterfactuals in knowledge-based programming. In I. Gilboa, editor, *Theoretical Aspects of Rationality and Knowledge, Proceedings of the Seventh Conference (TARK 1998)*, pages 97–110, San Francisco, California, July 1998. Morgan Kaufmann.

[15] 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.

[16] W. Janssen. Layers as knowledge transitions in the design of distributed systems. In U. H. Engberg, K. G. Larsen, and A. Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS* (Aarhus, Denmark, 19–20 May, 1995), number NS-95-2 in Notes Series, pages 304–318, Department of Computer Science, University of Aarhus, May 1995. BRICS.

[17] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

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

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

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

[21] Y. Moses. *Knowledge in a distributed environment*. PhD thesis, Stanford University, 1986.

[22] 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.

[23] 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.

[24] 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.

[25] R. van der Meyden and Y. Moses. On refinement and temporal annotations. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRTFT 2000 Pune, India, September 20–22, Proceedings*, volume 1926 of *LNCS*. Springer-Verlag, 2000.

[26] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, 1971.

A. Valid Refinement Rules from [25] and [7]

Let $P \asymp^{\mathcal{S}} Q$ denote the fact that both $P \leq_{\iota}^{\mathcal{S}} Q$ and $Q \leq_{\iota}^{\mathcal{S}} P$.

Reducing the amount of nondeterminism, introducing a coercion, and eliminating an assertion:

$$P \leq_{\iota}^{\mathcal{S}} P + Q \qquad [\phi]^X \leq_{\iota}^{\mathcal{S}} \varepsilon \qquad \varepsilon \leq_{\iota}^{\mathcal{S}} \{\phi\}_C$$

The empty program is a left and right identity for sequential composition.

$$\varepsilon * P \asymp^{\mathcal{S}} P \qquad P \asymp^{\mathcal{S}} P * \varepsilon$$

Axioms for iteration:

$$\varepsilon \leq_{\iota}^{\mathcal{S}} P^{\omega} \qquad P \leq_{\iota}^{\mathcal{S}} P^{\omega} \qquad P^{\omega} * P^{\omega} \leq_{\iota}^{\mathcal{S}} P^{\omega}$$

Sequential composition is assocative and distributes over nondeterministic choice.

$$(P * Q) * R \asymp^{\mathcal{S}} P * (Q * R)$$
$$(P + Q) * R \asymp^{\mathcal{S}} (P * R) + (Q * R) \qquad P * (Q + R) \asymp^{\mathcal{S}} (P * Q) + (P * R)$$

Nondeterministic choice is associative, commutative, and idempotent.

$$(P + Q) + R \asymp^{\mathcal{S}} P + (Q + R) \qquad P + Q \asymp^{\mathcal{S}} Q + P \qquad P + P \asymp^{\mathcal{S}} P$$

Call a program *vanishing* if it is the empty program, a coercion, or an assertion. Commutativity and Idempotence of vanishing programs: For all vanishing P, Q we have:

$$P * Q \asymp^{\mathcal{S}} Q * P \qquad P * P^{\omega} \asymp^{\mathcal{S}} P$$

Strengthen coercion:

$$[\phi \wedge \psi]^X \leq_{Y \hookrightarrow \{X\}}^{\mathcal{S}} [\phi]^Y$$

Strengthen precondition with coercion:

$$[\phi]^X [\phi \wedge \psi, \psi']^Y \leq_{Y \hookrightarrow \{X\}}^{\mathcal{S}} [\phi]^X [\psi, \psi']^Y \qquad \textbf{tighten-pre}$$

Loosen postcondition with coercion:

$$[\phi, \psi']^Y [\psi]^X \leq_{Y \hookrightarrow \{X\}}^{\mathcal{S}} [\phi, \psi]^Y \qquad \textbf{loosen-post}$$

Assertion introduction:

$$[\phi]^X \{\phi\}_{\{X\}} \leq_{\iota}^{\mathcal{S}} [\phi]^X$$

Coercion elimination, provided $X \notin C$:

$$\varepsilon \leq_{X \hookrightarrow C}^{\mathcal{S}} \{\phi\}_C [\phi]^X$$

Specification elimination, provided $X \notin C$:

$$\varepsilon \leq^{\mathcal{S}}_{X \hookrightarrow C} [\phi, \psi]^X \, \{\phi \to \psi\}_C \, [\phi]^X$$

Advance Box:

$$P * \{\Box\phi\}_C \leq^{\mathcal{S}}_{\iota} \{\Box\phi\}_C * P$$

Regress diamond:

$$\frac{P \leq^{\mathcal{S}}_{X \hookrightarrow C} [\mathit{true}, \mathit{true}]^X}{\{\Diamond\phi\}_{C \cup C'} * P \leq^{\mathcal{S}}_{\iota} P * \{\Diamond\phi\}_{C'}}$$

Strengthen specification:

$$\frac{\mathcal{S} \models \phi_1 \to \phi, \quad \mathcal{S} \models \psi \to \psi_1}{[\phi, \psi]^X \leq^{\mathcal{S}}_{Y \hookrightarrow \{X\}} [\phi_1, \psi_1]^Y}$$

Specification consequence:

$$[\phi, \psi]^X * \{\psi\}_{C \cup \{X\}} \leq^{\mathcal{S}}_{\iota} \{\phi\}_C * [\phi, \psi]^X$$

Trivial assertion:

$$\{\mathit{true}\}_{\emptyset} \leq^{\mathcal{S}}_{\iota} \varepsilon$$

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

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

$$\exists_i p\left([\phi]^X\right) \leq^{\mathcal{S}}_{\iota} [\exists_i p\,(\phi)]^X \qquad \textbf{ext-lq}$$

$$\exists_i p\left(R[{}^{P}/_{Z}]\right) \leq^{\mathcal{S}}_{\iota} R[{}^{\exists_i p(P)}/_{Z}] \quad \textit{if } p \textit{ not free in } R \qquad \textbf{lift-lq}$$

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.

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