Issues for a Temporal Refinement Calculus

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

[1] Computing Sciences
University of Technology, Sydney
P.O. Box 123, Broadway,
NSW 2007, Australia
ron@socs.uts.edu.au
http://www-staff.socs.uts.edu.au/~ron
[2] Electrical Engineering
Technion, 32000
Israel
moses@ee.technion.ac.il

Abstract. This note discusses a number of issues that arise in the development of a refinement calculus for a programming language in which specification statements may involve temporal logic statements. We consider a sequential (i.e., non-distributed) setting and aim at a *linear-time* semantics for the temporal logic. The added power due to the use of temporal operators is illustrated, and subtleties having to do with proving assertions *about* the program and the use of labels in a program are considered. This work is part of a project whose goal is to provide a refinement calculus for distributed programs using a rich specification language.

1 Introduction

We are engaged in a project[1] to develop a formal framework for the design of distributed algorithms. Our starting point in this work was an approach to the analysis of such algorithms that is based on logics of knowledge [HM90], which has been shown in a number of works in the last decade to provide unique insights into the nature of distributed computation. The *knowledge-based programs* of Fagin et al [FHMV95] provide a formal syntax and semantics within which this approach can be represented. Knowledge-based programs resemble ordinary programs in their syntax, but they are better viewed as constituting abstract *specifications* of the way a process' concrete actions relate to its state of knowledge. *Knowledge-oriented programs* [MK93] take the level of abstraction one step further, allowing abstract actions such as `notify` whose purpose is to effect changes in processes states of knowledge.

We aim to generalize the frameworks of knowledge-based and knowledge-oriented programs. Inasmuch as these are a hybrid of program and specifications,

[1] Supported by an ARC Large grant and by a grant from the Israeli Ministry of Science.

they resemble *refinement calculi*, which similarly view programs and specifications as belonging to the same semantic category. This analogy has led us to the investigation of refinement calculi for distributed systems that are capable of representing knowledge-based programs. The work has required us to generalize and amend previous approaches to program refinement along several dimensions. Specifications of and reasoning about distributed computation deal inherently with temporal behaviour. Thus, one of the issues we have found it necessary to consider is how to integrate program refinement and temporal logic. The purpose of this note is to briefly describe our approach to such an integration (in the context of non-distributed programs, for which the problem is already quite complex) and some of the issues we have had to face. Details will be provided in a forthcoming paper [MM98a]. Other aspects of our larger program of research are treated in [EvdMM98,MM98b].

Previous work on program refinement can be categorised into two classes. One of the most deeply explored approaches [Mor90a,BvW98] is *state-based*, being premised on the use of predicate transformers and weakest preconditions as a semantic basis. This is natural for programs whose specifications are descriptions of input/output relations, of which predicate transformers are a generalization. One of the advantages of this approach is the ability to write specifications containing two distinct types of annotation: one type of annotation is used to state properties that the program is *required* to satisfy, the other is more like conventional program annotations in that it is used to state properties that the program has been *proved* to satisfy.

There have been proposals to extend the application of the predicate transformer based approach to distributed computations, by viewing a distributed program as a predicate transformer which is repeatedly applied [Bac90,BS91]. (There are also closely related proposals to base the semantics of temporal logic on predicate transformers [Mor90b,LvdS92].) A disadvantage of this idea is that the rich compositional structure of the standard refinement calculus is lost.

Another class of refinement calculi is *action-based* [GS86,Hol89,HL95,Win86]. Work in this category begins with a process calculus and enriches it with features of modal logic, typically Henessy-Milner logic or the more expressive μ-calculus. This approach has the benefit of allowing greater compositional freedom. On the other hand, these works do not admit the two types of annotation available in the state-based approaches.

The contribution of our work can be viewed as being to develop a framework that has both the rich compositional structure and modal logic expressiveness of the process calculus based approaches and the ability to represent the two types of annotation available in the predicate transformer based approaches to refinement of sequential programs. Instead of an action based modal logic, we use temporal logic. In Section 2 we sketch a program calculus based on temporal constraints. Section 3 extends this calculus with assertions that capture annotations by proved properties. Section 4 deals with a further useful enhancement, the use of labels that may be referred to by the annotations.

2 Programs with Temporal Constraints

Our approach is grounded in the standard semantics of temporal logic [MP92]. Suppose for the sake of discussion that we are interested in a propositional temporal logic in which for every formula φ there is a formula $\Diamond\varphi$, intuitively asserting that φ will eventually be true. The basic semantic construct in the standard semantics is a *run*, a mapping from timepoints (the natural numbers $\mathbf{N}$, say) to states of the system under consideration. A *point* is a tuple (r, n) where r is a run and n is a natural number. We may define the relation "$\models$" of satisfaction of a formula at a point by

1. when p is an atomic proposition, $(r, n) \models p$ if p holds in the state $r(n)$,
2. $(r, n) \models \Diamond\varphi$ if $(r, m) \models \varphi$ for some $m \geq n$.

This is the *linear time* semantics in the style of Manna and Pnueli [MP92]. As usual, we write $\Box\varphi$ for the dual $\neg\Diamond\neg\varphi$, which asserts that φ is true at all future times.

In order to develop a theory of program refinement, we start by defining three elements:

- A language of programs.
- An *executes* relation $\Vdash$ between intervals of runs and programs. An interval is a triple $r[n, m]$ where r is a run and $n \leq m \leq \infty$ are elements of $\mathbf{N} \cup \{\infty\}$. Intuitively, $r[n, m] \Vdash P$ holds when the program P executes from time n to time m in the run r.
- A refinement relation $\sqsubseteq$, where we write $P \sqsubseteq Q$ for *program P refines program Q*.[2]

Our language of programs contains analogues of the usual constructs for sequential programs. Chief amongst these is an operator $*$ which resembles sequential composition. Its semantics is given by $r[m, n] \Vdash P * Q$ if there exists m' such that $r[m, m'] \Vdash P$ and $r[m', n] \Vdash Q$.

In addition to standard operators of a sequential language, our language of programs contains two types of basic programs that are thought of as *constraints*, which are in the spirit of similar constructs used by Morgan [Mor90a] and by Back and von Wright [BvW98]. The first are the so-called *coercions*, which have the form $[\varphi]$ and force φ to hold. The second are *specifications*, which have the form $[\varphi, \psi]$, standing for a program that, if started at a point satisfying (the *initial condition*) φ, will terminate in a point satisfying (the *goal*) ψ. Roughly speaking, we want the $\Vdash$ relation for such program constructs to satisfy

- $r[n, m] \Vdash [\varphi]$ if $n = m$ and, if $n < \infty$, then $(r, n) \models \varphi$; and
- $r[n, m] \Vdash [\varphi, \psi]$ if either
  1. $n = m = \infty$ or
  2. $n < \infty$ and $(r, n) \not\models \varphi$ or

[2] We use the reverse of the order preferred by many authors, for reasons having to do with correspondence to our semantics of this relation.

3. $n \leq m < \infty$ and $(r, n) \models \varphi$ and $(r, m) \models \psi$.

By being instances of basic programs, constraints can be composed with other constructs, resulting in composite programs that can have elements which are programs at different levels of abstraction.

Allowing temporal operators in the formulas enables the coercions and specifications to talk about ongoing and reactive behaviors. This gives our framework an expressive power that goes beyond input/output relations. For example, we may write programs such as

$$[\Box(x < 10)] * [\Diamond(x = 3), x = 2]$$

This specifies a program that ensures that the value of x is always less than 10, and if started in a situation where it is guaranteed that eventually x equals 3, will terminate with x equal to 2.

Given the elements described above, we can define a refinement relation on programs by $P \sqsubseteq Q$ if for every $r[n, m]$, if $r[n, m] \Vdash P$ then $r[n, m] \Vdash Q$. This captures the intuition that refinement corresponds to the addition of constraints to the program, and the reduction of any nondeterministic behaviour it may allow. While this definition is intuitive, however, we have not yet captured some important features of the standard refinement calculi, that make a significant contribution to their practicality.

3 Assertions

It is often very useful to be able to reason about a partially refined program, some of whose components may still involve constraints. Properties guaranteed by certain parts of the program can then be made use of to improve the refinement of other parts of the program. This becomes especially useful once our formulas involve temporal operators, because then it may be possible to infer temporal properties at one point from those that are proven to hold at another. We call such conclusions about the program *assertions*, and denote the assertion that φ holds by $\{\varphi\}$. Syntactically, assertions are very similar to coercions. Semantically, however, they are treated differently.

One of the features we wish to make use of is the fact that assertions may be stated based on constraints in the program, which we think of as program fragments that will be developed later in the refinement process. For example, at a location in the program where a coercion $[\varphi]$ appears, we may soundly assert that φ must hold. Every concrete program that refines the current text will indeed satisfy φ at that location. Conversely, one way to demonstrate that a coercion has been satisfied is by proving that the formula being coerced is a sound assertion there. Done properly, this should be a way of eliminating coercions in the refinement process.

The generation and use of assertions involves a number of subtle points. First, it is essential that the two operations mentioned above: introducing an assertion based on a coercion, and eliminating a coercion based on an assertion, are not applied inappropriately, as in the following derivation:

$$\epsilon \sqsubseteq \{\varphi\} \sqsubseteq \{\varphi\} * [\varphi] \sqsubseteq [\varphi]$$

Here ϵ stands for the empty program which does nothing and takes no time to execute. The derivation should be read from right to left. In its first step, the assertion $\{\varphi\}$ is concluded from the coercion $[\varphi]$.[3] In the second step, the same assertion is used as basis for eliminating the coercion(!) The third step eliminates the assertion, which is a step that is consistent with our view of assertions as stating properties that have already been established, and therefore not constraining the refinement process. The problem in this example is, of course, the circularity of the reasoning. Similar issues can arise in much more subtle ways, especially in the presence of temporal formulas. For example,

$$\epsilon \sqsubseteq [\Diamond p, \Diamond p] \sqsubseteq \{\Diamond p\} * [\mathbf{true}, \Diamond p] \sqsubseteq [\mathbf{true}, \Diamond p] * \{\Diamond p\} \sqsubseteq [\mathbf{true}, \Diamond p]$$

is a problematic derivation due to a similar circularity.

In order to avoid such pitfalls, our formalism is enhanced to keep track of the constraints on which the statement of an assertion is based. Technically, we label constraints with *constraint variables* from a set CV, so that a coercion, say, will now have the form $[\varphi]^X$ where $X \in CV$, and every assertion appears as $\{\psi\}_J$ where $J \subseteq CV$ is considered the *justification set* for this assertion. Intuitively, an occurrence of $\{\varphi\}_J$ at a location in a program states that φ holds at this location, with the proof depending only on constraints labelled by variables in J. We now allow the introduction of assertions as in

$$\{\varphi\}_{\{X\}} * [\varphi]^X \sqsubseteq [\varphi]^X.$$

We also have rules for the elimination of constraints, such as

$$\epsilon \sqsubseteq \{\varphi\}_J * [\varphi]^X \qquad \text{if } X \notin J.$$

The restriction in this rule suffices to block the first problematic derivation above, and similar restrictions can guarantee to block the second one. Properly keeping track of the reasoning underlying the assertions in this way requires, however, a somewhat more complex semantics for both programs and refinement. We need to guarantee that refinement steps generate correct assertions, and may need to modify justification sets in the course of a refinement step. The definition of refinement we use is a generalization of that of the previous section. The details are beyond the scope of this brief abstract (but see [MM98b]).

[3] It may be surprising at first to see the assertion placed before the assertion, when intuitively it is “caused” by the coercion. This is not an error, however, since the assertion and the coercion refer to the same point of the run, and therefore commute.

4 Labels

In order to translate reasoning between the program constructs and the temporal logic, we have found it useful to introduce labels as a way of marking specific locations in a program to which the temporal assertions may refer. Intuitively, to each label l there exists an atomic proposition $\mathtt{at}(l)$ true just at the locations marked by l in the program. Labelled programs and reasoning using the propositions $\mathtt{at}(l)$ are a central feature of the approach to temporal verification of Manna and Pnueli [MP92,MP95]. In this approach, the whole program needs to be translated to a formula of temporal logic. The incorporation of labels in our framework makes it possible for us to simulate this approach. However, because we have a richer syntax, we have greater flexibility to do reasoning based directly on the structure of the program, without having to translate it to temporal logic in its entirety.

Labels also add to the expressive power of our refinement calculus in interesting ways. For example, provided the label l does not occur in P,

$$\exists l([\Diamond \mathtt{at}(l)] * P * l)$$

defines a program whose behaviours are precisely the *terminating* behaviours of the program P. Here the existential quantifier functions as a scoping (or hiding) operator whose purpose is to delimit the points of the run at which l occurs to those referred to by the program in its scope.

5 Conclusion

The framework we have sketched above is able to capture much of the reasoning that can be represented in refinement calculi for sequential programs based on predicate transformers, but not all. Restricting our framework to sequential programs, with assertions restricted to be about states, would make its semantic basis be roughly equivalent to the set of relations over a given collection of states. The set of predicate transformers forms a much larger semantic space. This allows standard refinement calculi to represent "miraculous" programs which, while they cannot correspond to any concrete program as representable in a conventional programming language, can nevertheless be used during program derivation [Mor88]. Our framework does not admit such miraculous programs. Since we do have the capacity to handle standard temporal reasoning in our framework, we believe the price is one worth paying.

References

[Bac90] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[BS91] R.J.R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.

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

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

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

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

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

[HM90] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990. A preliminary version appeared in *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984.

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

[LvdS92] J.J. Lukkien and J.L.A. van de Snepscheut. Weakest preconditions for progress. *Formal Aspects of Computing*, 4:195–236, 1992.

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

[MM98a] R. van der Meyden and Y. Moses. A temporal refinement calculus. forthcoming, 1998.

[MM98b] R. van der Meyden and Y. Moses. Top-down considerations on distributed systems. In *International Symposium on Distributed Computing*, Andros, Greece, Sept 1998. To appear.

[Mor88] C. Morgan. Data refinement using miracles. *Information Processing Letters*, 26:243–246, 1988. Jan.

[Mor90a] C. Morgan. *Programming from Specifications.* Prentice Hall, New York, 1990.

[Mor90b] J. M. Morris. Temporal predicate transformers and fair termination. *Acta Informatica*, 27:287–313, 1990.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, 1992.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems.* Springer-Verlag, New York, 1995.

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