Total-Correctness Refinement for Sequential Reactive Systems*

Paul B. Jackson **

Division of Informatics
University of Edinburgh
Edinburgh EH9 3JZ, UK
pbj@dcs.ed.ac.uk

Abstract. We introduce a coinductively-defined refinement relation on sequential non-deterministic reactive systems that guarantees total correctness. It allows the more refined system to both have less non-determinism in its outputs and to accept more inputs than the less refined system. Data reification in VDM is a special case of this refinement.
Systems are considered at what we have called *fine* and *medium* levels of granularity. At the fine-grain level, a system's internal computational steps are described. The fine-grain level abstracts to a medium-grain level where only input/output and termination behaviour is described. The refinement relation applies to medium grain systems.
The main technical result of the paper is the proof that refinement is respected by contexts constructed from fine grain systems. In other words, we show that refinement is a precongruence.
The development has been mechanized in PVS to support its use in case studies.

1 Introduction

Refinement. Refinement is a fundamental verification methodology and has a strong conceptual appeal. It takes a black-box view of systems, characterizing them by their observable interface behaviour.

Let A be an abstract system, C a concrete system, and assume that it has been shown that A *refines to* C, written $A \sqsubseteq C$. A good definition of $\sqsubseteq$ and theory of refinement then provide a guarantee that we can substitute C for A in any environment with no observable consequences.

Formally one way to give evidence for substitutivity is to show a *precongruence* property:

$$\vdash A \sqsubseteq C \Rightarrow \mathcal{E}[A] \sqsubseteq \mathcal{E}[C]$$

 In proceedings of TPHOLs 2000, *13th International Conference on Theorem Proving in Higher Order Logics*, J. Harrison and M. Aagaard editors, volume 1869 of Lecture Notes in Computer Science, pages 320-337. Springer-Verlag, August 2000.

** Also affiliated with the Institute for System Level Integration, Livingston, UK

of $\sqsubseteq$ for a class of contexts or environments $\mathcal{E}$.

This paper introduces a new definition of a refines-to relation for sequential non-deterministic systems that addresses weaknesses of previously proposed relations. The main technical result is to prove this refines-to relation to be a precongruence for a general class of environments.

Inclusion-Based Refinement. Many common definitions of a refinement relation involve inclusion. For example refinement might assert that a step transition relation of the concrete system is included in that of the abstract system, or that every trace of the concrete system is also a trace of the abstract system, a *trace* being a sequence of observable states or input/output values. Such definitions have several problems, as we explain in the next two subsections. We use trace inclusion as an example, but our remarks apply to other inclusion-based definitions too.

Contravariance of Inputs. A consequence of a trace-inclusion definition is that, if there is some step of behaviour in the concrete trace corresponding to the environment passing the system some input, there must be a similar step in the abstract trace. This is intuitively the wrong way round and allows a bad concrete system to inadvertently constrain environment behaviour and falsely appear to be correct. A variety of approaches have tried to deal with this. For example, the notion of *receptivity* is introduced [5].

Total Correctness. We consider it important that a refinement relation capture total correctness. Without totality, it is much harder to argue that a concrete system could replace an abstract system with no observable consequences.

Trace inclusion is a partial correctness rather than total correctness notion. It requires that when the concrete system makes some step of behaviour passing output to the environment, the abstract system must make some matching step. However it doesn't ever require that the concrete system make any output step in the first place.

As explained in [4], one adaptation for total correctness is to introduce an extra value $\perp$ into the state spaces of systems. If a system originally is not guaranteed to make an output step from some given state, a transition to $\perp$ is added, along with a transition to every other state. There is therefore no possibility for a system to be blocked from making a step, all systems are total. Furthermore, on starting from a $\perp$ state, a system must non-deterministically be able to transition to every possible state (including $\perp$ again). With this setup, trace inclusion requires that, whenever the abstract system is capable of making only controlled steps (i.e. not to a $\perp$ state), the concrete system also can only make controlled steps, and so inclusion now enforces total correctness.

The Proposed Refines-to Relation. We propose a *refines-to* relation that directly captures the desirable contravariant relationship between inputs of abstract and concrete systems, and that ensures total correctness without the complications of adding extra $\perp$ states. We define refines-to coinductively. See Sec. 4 for details.

Fine and Medium Grain Systems. We focus our attention on non-deterministic sequential systems that alternately accept an input value from some environment and return an output value back to the environment. We assume systems can modify some internal state and that this state is preserved between returning an output value and accepting some next input.

We use an automata-based *fine-grain* model for describing system implementations. See Sec. 5. This model represents atomic computation steps and can exhibit phenomena such as divergence and deadlock. Fine grain systems abstract to a *medium-grain* level where just the input/output and termination behaviour of systems is captured. The medium grain level is also appropriate for directly creating system specifications. See Sec. 3 for the medium grain system definition. This medium grain formalism uses precondition and transition relations and is very similar to the way systems are described in VDM [10], for example.

Refines-to is defined only on medium grain systems. It is independent of how we characterise systems at the fine grain. For example, for the fine grain model we could have used instead a structured operational semantics that captures total correctness. One advantage of an automata-based approach to fine grain systems is that the characterisation of when systems terminate is direct and obviously correct.

When showing that *refines-to* is a precongruence, we use a variation on fine-grain systems to construct the general class of environments that we show precongruence with respect to. See Sec. 6 for the definition of the variation and Sec. 7 for the precongruence proof.

Evaluating Goodness of Refines-to. A precongruence property is generally desirable for any refinement relation, but isn't sufficient by itself to justify the relation's definition. To take an extreme example, an always true refinement relation is indeed a precongruence, it but provides no substitutivity guarantees at all. We also must look at the environment beyond the boundaries of the system we model formally, and consider the expectations this environment has.

Sometimes, for example in the process algebra community, these expectations are formalized by developing a theory of *testing* [6] and showing (at least) that any more refined system passes all tests that a more abstract system passes. The hope is that it is more straightforward to agree that a testing theory adequately captures the expectations of an external environment than to agree that the refinement relation does.

We haven't developed a testing theory, and instead simply discuss the expectations we might reasonably have of sequential reactive systems. We do observe that the total-correctness proof obligations adopted in VDM for showing that one sequential program is a data reification of another are a consequence of our definition of *refines-to*. Also, it would be easy to derive the similar VDM obligations for showing an implementation meets a specification.

Use of a Theorem Proving System. We see all of the PVS [13] formalization work described in this paper as being necessary support material for case studies in verifying actual systems.

It's worth mentioning too that we also found the use of PVS a significant help in clarifying what definitions were necessary, how lemmas should be phrased, and how proofs should go. At the same time, we found the main proofs sufficiently intricate and the weight in the formal notation sufficiently high that in many cases it was necessary to be sketching proofs on paper before or at the same time as attempting the PVS proofs.

An Illustrative Example. We show in Sec. 8 a specification of an abstract data type of sets as a medium grain system, and an implementation as a fine grain system.

2 Related Work

The use of coinduction to define refinement relations has been made popular by the process algebra community [12].

Jacobs in [9] characterises classes in object-oriented languages as coalgebraic categories, and uses a coinductive notion of refinement to specify correctness of implementations. His approach is more general than ours in that he allows for changes in the system interface in going from abstract to concrete. However, he takes a simpler view of systems: he models them using total functions so non-determinacy is not possible, and he doesn't take account of any input preconditions that might need to be satisfied for termination. We imagine it would be possible to adapt these extra features that we consider into his framework. This work is also being implemented in PVS.

We originally considered a coinductive definition of a refinement relation that allows contravariance on inputs after seeing Abramsky discuss such a relation [1]. The relation he considers is on labelled transition systems with input and output labels on the transitions. He also has a game-theoretic version that applies to prefix-closed sequences of input/output behaviour. One limitation of his relation is that it captures partial, not total correctness.

A formalism for concurrent systems that allows contravariance on inputs and prevents restriction of environment behaviour by the system is that of alternating refinement relations [3]. This work also uses coinductive characterisation of refinement. To tackle concurrency issues, its definition is more elaborate than ours. For example, the nesting depth of alternations of quantifiers is four, compared to 2 in our case. In the reactive modules [2] formalism being pursued by a subset of the authors of [3], a notion of *temporal abstraction* is defined, much like our map from fine to medium grain systems, that can hide internal computation steps of system components.

In the literature on refinement of sequential programs (see [4] for a recent comprehensive survey), our approach is closest to that taken in VDM [10]. Our

medium grain systems exactly correspond to the precondition and VDM post condition[1] style specifications.

Early work of Milner [11] looks at denotational semantics for *transducers* which are effectively the same as our medium grain systems. To our knowledge, Milner never proposed a coinductive definition of refinement for transducers, though he deployed coinductive definitions heavily in his concurrency theory work on labelled transition systems. There the distinction between inputs and outputs is erased at the level much of the semantics work is carried out, so the opportunity we take to treat them differently is lost.

3 Medium Grain Systems

A *medium grain system* is a full description of the behaviour of a non-deterministic sequential reactive system from an input/output and termination point of view. The intent is that both system specifications and implementations can be phrased as medium grain systems.

The type `Med_gs` of medium grain systems , parameterized by types `I` and `O` of input and output values and type `Q` of internal states, is defined as a subtype of a record type:

```
Med_gs[Q,I,O] : TYPE ≐
 {s : ⟨ pre ⊆ Q × I,
        trans ⊆ Q × I × O × Q   ⟩
 |
  ∀p,i. s.pre(p,i) ⇒ ∃q,o. s.trans(p,i,o,q)  }.
```

The notation '`fieldname ⊆ Type`' abbreviates '`fieldname : 𝒫(Type)`' where $\mathcal{P}$ is the powerset (set of subsets) operator. Subsets of a type `T` are represented as functions of type `T → bool`, so membership of an element `x` in a subset `s` is expressed as function application `s(x)`, a notation in keeping with the correspondence between subsets and predicates.

The field `trans` specifies what transitions the system can make. The relation `trans(p,i,o,q)` indicates that, starting from state `p` and presented with input value `i`, it is possible for the internal computations of the system to eventually terminate in state `q` and for the system to return output value `o`. Because of non-determinism there might be more than one `q` and `o` for given `p` and `i`. The field `pre` specifies a precondition. The relation `pre(p,i)` indicates that starting from state `p` and presented with input `i`, the internal computations of the system are guaranteed to terminate in some state from which output is generated. In general it is not equivalent to `∃q,o. trans(p,i,o,q)` but stronger. Even if a system can reach `q` and output `o` from a given state `p` and input `i`, because of non-determinism, it might also deadlock or go into a divergent computation.

[1] relations on inputs and outputs, rather than just relations on outputs as in Hoare style specifications

It would be convenient to include the type parameter `Q` as an initial field of the record type in the definition of `Med_gs`. However this is not possible in the PVS specification language.

To fully describe in PVS a medium grain system, we sometimes augment the presentation of the system as an element of `Med_gs` by identifying some element of `Q` as the system's initial state.

We imagine interactions between an environment and a medium grain system as a continuing dialogue: if the thread of control is with the environment, the environment can choose to provide the system with some input. The system then processes this input and possibly eventually generates some output. Control then passes back to the environment which is free is choose some further input for the system.

For some purposes, we make the assumption that the environment has no ability to access or modify the internal system state. The environment might only know that the system initially started off in some well-characterized state.

We imagine that a medium grain system being used as a specification will exhibit a range of possible behaviour on a given input that is only dependent on the initial state and the observed input/output behaviour inbetween. We consider a system with this property to be *coarse grain*. Coarse grainness is a desirable property for specifications. Coarse grainness corresponds to determinacy in the CCS process calculus [12]. We haven't yet made any use of coarse grainness in our work.

4 Definition of Refinement

Our definition of what it means for one system to be a refinement of another is in the style of the coinductive definition of bisimulation [12].

Fix on an abstract medium grain system `sa` and a concrete medium grain system `sc` with distinct internal states `Qa` and `Qc` and both over input type `I` and output type `O`: in PVS, they have respective types `Med_gs[Qa,I,O]` and `Med_gs[Qc,I,O]`.

A relation `R` $\subseteq$ `Qa` $\times$ `Qc` is a *refinement relation* from `sa` to `sc` iff it satisfies

```
R(pa,pc) ⇒                                                    (1)
  ∀i. sa.pre(pa,i) ⇒
    sc.pre(pc,i)
    ∧ ∀qc, o. sc.trans(pc,i,o,qc) ⇒
          ∃qa. sa.trans(pa,i,o,qa) ∧ R(qa,qc)
```

for any states `pa` and `pc`.

System `sa` in initial state `inita` *refines to* system `sc` in initial state `initc`, written

```
refines_to(sa,sc)(inita,initc),
```

iff there exists a refinement relation `R` such that `R(inita,initc)`. The relation `refines_to(sa,sc)` is easily shown itself to be a refinement relation, and so by

this definition it is the greatest refinement relation, adopting the usual ordering of relations by inclusion.

We realise the definition of `refines_to` in PVS as the greatest fixed point of the appropriate functional. PVS doesn't provide direct support for such coinductive definitions. However, we easily prove a lattice-theoretic version of the Tarski-Knaster fixed-point theorem and specialise it for the creation of coinductive definitions over the lattice of subsets of a type.

Trivially we show that `refines_to` is a preorder, that is, it is reflexive and transitive.

Why is this definition plausible? Assume we have found a refinement relation `R`, system `sa` is in some state `pa`, system `sc` is in some state `pc`, and `R(pa,pc)` holds. We then know for a start that

```
∀i. sa.pre(pa,i) ⇒ sc.pre(pc,i).                                    (2)
```

This is very reasonable: system `sc` is guaranteed to converge to an output on every input that `sa` converges on. System `sc` might converge also on other inputs too, but that doesn't matter here.

We also know

```
∀i, qc, o. sa.pre(pa,i) ∧ sc.trans(pc,i,o,qc) ⇒                     (3)
    ∃qa : sa.trans(pa,i,o,qa) ∧ R(qa,qc).
```

Any output that the concrete system generates on an abstractly acceptable input is also an abstractly acceptable output. The concrete system's output behaviour is always what we might expect. The concrete system might exhibit less non-determinism, completely in line with the approach in specification of introducing non-determinism, not because that non-determinism is expected in any one implementation, but in order to permit flexibility in implementation. However (2) guarantees that there always is *some* output that the concrete system generates. Importantly too from (3) we know `R(qa,qc)` holds, so we also expect all subsequent I/O behaviour of the concrete system to be in accordance with the abstract system behaviour.

We make the assumption above that an environment would never want to supply a system with input when there is not a firm expectation that the system will eventually generate some output given that input. We are not trying to define a notion of refinement that is to be used when thinking about the fault tolerance of systems or about systems that have divergent computations in the normal course of events.

Having said that, this definition of refinement should also be applicable if only partial correctness were of interest. There is nothing intrinsic in the definition itself that refers to total correctness. However for partial correctness one would want to discard the subtyping condition we have used in the definition of the `Med_gs` type that requires at least one output value to exist whenever the precondition is satisfied.

A common approach to establishing a refinement relationship between an abstract and concrete system involves introducing a function `rmap` of type `Qc`

$\longrightarrow$ `Qa` (sometimes known as a *refinement mapping*, *abstraction map* or *retrieve function*) and an invariant on concrete states `c_inv` $\subseteq$ `Qc`. The function `rmap` and predicate `c_inv` define a refinement relation:

```
R(pa,pc) : bool ≐ c_inv(pc) ∧ pa = rmap(pc).
```

Specialising (1), a predicate stating that `rmap` and `c_inv` form a refinement relation is

```
refmap_step(sa,sc,c_inv,rmap) : bool ≐
   ∀pc,i.
     c_inv(pc) ∧ sa.pre(rmap(pc),i) ⇒
       sc.pre(pc,i)
       ∧ ∀qc, o. sc.trans(pc,i,o,qc) ⇒
             sa.trans(rmap(pc),i,o,rmap(qc)) ∧ c_inv(qc),
```

and the coinduction principle that goes with `refines_to` specialises to the theorem `refines_to_ind_with_refmap_a`:

```
⊢ c_inv(initc) ∧ inita = rmap(initc)
   ∧ refmap_step(sa,sc,c_inv,rmap)
   ⇒
      refines_to(sa,sc)(inita,initc).
```

We observe that the antecedents of this theorem are exactly a strict subset of the proof obligations in VDM [10] for establishing a data reification relationship between an abstract data type and its implementation when there is also an invariant on the concrete type.

The extra proof obligation in [10] concerns *adequacy*. In our notation:

```
∀qa. ∃ qc. c_inv(qc) ∧ qa = rmap(qc).
```

This is usually desirable because it says that every abstract value has at least one concrete representation. However it is not necessary for showing

```
refines_to(sa,sc)(inita,initc).
```

If it so happens that in the abstract system `sa` starting from state `inita` we cannot access every abstract state by some sequence of input values, then adequacy needn't hold for the inaccessible states. This is unlikely to happen if the abstract system is an initial specification, but it could reasonably happen if it is a system at some intermediate level of refinement.

We also observe that if the preconditions `sa.pre` and `sc.pre` in `refmap_step` are always true, the theorem `refines_to_ind_with_refmap_a` reduces to the induction principle commonly used when refinement is defined as trace inclusion.

5 Fine Grain Systems

5.1 Definition of Fine Grain System

A *fine grain system* is a system description that allows the presentation of the individual computation steps that a system can perform. It is a suitable formalism for describing system implementations: see Sec. 8 where an example is given of describing the procedures in an imperative implementation of an abstract data type as a fine grain system.

We build on the definition of a fine grain system when defining later the contexts or environments that some medium grain system may be operating in. See Sec. 6 and Sec. 7.

We define a type `Fin_gs` of fine grain systems as

```
Fin_gs[Q,I,O] : TYPE ≐
 {s : ⟨ run ⊆ Q,
        input : (Q × I) ⟶ Q,
        step ⊆ Q × Q,
        output : Q ⟶ O,
        wbehaved ⊆ Q ⟩
  |
    ∀p,q. s.step(p,q) ⇒ s.run(p)  }
```

with parameters `Q`, `I`, and `O` as in the definition of the `Med_gs` type in Sec. 3.

Initially a fine grain system is in some state for which `run` is false. When an input value is presented to a fine grain system, the system uses `input` to transition to a new state. The system then uses `step` to repeatedly make nondeterministic internal transitions. As specified by the subtyping predicate, steps can only be taken from states that satisfy `run`. The system halts and uses `output` to generate an output value if and when it reaches a state for which `run` is false.

A system can *deadlock*, reach a state `p` for which `run` is true, but ¬∃`q. step(p,q)`. Deadlock might seem an unusual feature to have in a model of a sequential system, but it is a natural phenomenon for guarded transition systems to exhibit. Deadlock is one appropriate behaviour for handling exceptional situations without extra machinery in the formalism. And checks for its absence can reveal bugs in system descriptions.

A system can also *diverge*, perform steps ad-infinitum without ever reaching a state in which `run` is false.

Once halted, a system is then ready to be reactivated by a further input. The predicate `wbehaved` identifies those states from which any step by `step` is guaranteed to be well-behaved. A step might not be well-behaved if it involves interacting with a subsystem.

As with medium grain systems, to fully specify a fine grain system we often also identify some element of `Q` as the system's initial state.

5.2 Abstraction from Fine to Medium Grain

To form the input/output medium-grain view of a fine grain system `s` with type `Fin_gs[Q,I,O]`, we use the fine to medium grain map:

```
map_fm(s) : Med_gs[Q,I,O] ≐
   ⟨ pre := mgs_pre(s),
     trans := mgs_trans(s)
   ⟩,
```

where

```
mgs_pre(s)(p,i) : bool ≐
  progressive?(s)(s.input(p,i))
  ∧ ¬inf_chain(s.step)(s.input(p,i))

mgs_trans(s)(p,i,o,q) : bool ≐
  star(s.step)(s.input(p,i),q)
  ∧ ¬s.run(q)
  ∧ o = s.output(q)

at_progressive?(s)(q) : bool ≐
  s.run(q) ⇒ s.wbehaved(q) ∧ ∃r. s.step(q,r)

progressive?(s)(q) : bool ≐
  ∀r. star(s.step)(q,r) ⇒ at_progressive?(s)(r).
```

Here we draw on an auxiliary development of properties of finite and infinite sequences of values where adjacent values are related by a binary relation `R`. The relation `star(R)` is the reflexive transitive closure of `R`. The predicate instance `inf_chain(R)(x)` indicates that there exists an infinite chain of `R`-linked values starting from `x`. If `star(s.step)(q,r)`, then by the subtype property of fine grain systems, every state on any path from `q` up to but excluding `r` is a `run` state. A state is `progressive?` if every `run` state accessible by stepping through only `run` states is both well behaved and not deadlocked. The predicate name `at_progressive?` is an abbreviation for 'atomically progressive?'. The predicate `mgs_pre` identifies exactly those inputs of the fine grain system for which no divergence is possible and it is guaranteed that the system will eventually reach a halting state. It might be difficult when reasoning with actual systems to work with this definition of `mgs_pre`, and simpler to use instead some predicate known to be stronger than that given here. The predicate `mgs_trans` specifies what outputs the fine grain system might generate for each input.

With the typing of the `map_fm` definition, the PVS type checker automatically generates a TCC (type correctness condition) that requires us to check that the subtype predicate in the `Med_gs` definition is satisfied.

6 Parameterized Fine Grain Systems

6.1 Definition of Parameterized Fine Grain System

A *parameterized fine grain system* is an adaptation of a fine grain system that can feed inputs to and receive outputs from a medium grain subsystem. The use of it in this paper is as a general description of contexts that medium grain systems might operate in. The type of parameterized fine grain systems is:

```
Prm_fin_gs[Q,I,O,Ix,Ox] : TYPE ≐
 {s : ⟨  run ⊆ Q,
         input : (Q × I) ⟶ Q,
         output : Q ⟶ O,
         i_step ⊆ Q × Q,
         x_en ⊆ Q,
         x_input : x_en ⟶ Ix,
         x_output : (Q × Ox) ⟶ Q  ⟩
  |
    ∀p. s.x_en(p) ∨ (∃q. s.i_step(p,q)) ⇒ s.run(p)  },
```

where the type parameters for `Prm_fin_gs` are `Q` for internal states, `I` for values input from the environment, `O` for values output to environment, `Ix` for values fed to the medium grain subsystem, and `Ox` for values received back from the subsystem. The fields `run`, `input` and `output` are as for a fine grain system. The relation `i_step` is for internal steps, and the predicate `x_en`, function `x_input` and function `x_output` are for the interface to the subsystem. Their use will become clear in the next subsection.

6.2 Instantiation of Parameterized Fine Grain System

Here we show how to combine a parameterized fine grain system `s` with a medium grain system `x` to create an unparameterized fine grain system. There are several options as to how the internal state spaces of `s` and `x` are related. In general they might share some state and also each have some distinct private state. Our immediate interest is in the situation where the only interaction can be via `x`'s input/output interface, so we choose to keep the state spaces distinct. Let `Q` be the state space of `s` and `Qx` the state space of `x`. The type of states for the combined system is `Q × Qx`.

Let the type parameters `I`, `O`, `Ix`, and `Ox` be defined as in Sec. 6.1. The parameterised fine grain system `s` then has type `Prm_fin_gs[Q,I,O,Ix,Ox]` and the medium grain system `x` has type `Med_gs[Qx,Ix,Ox]`. The map instantiating `s` with subsystem `x` has definition:

```
m_ipfd(s, x) : Fin_gs[(Q × Qx),I,O] ≐
  ⟨ run   := λ(q,qx). s.run(q),
    input := m_ipfd_input(s,x),
    step  := m_ipfd_step(s,x),
```

```
    output := λ(q,qx). s.output(q),
    wbehaved := m_ipfd_wbehaved(s,x)   ⟩,
```

where

```
m_ipfd_step(s, x)(ppx,qqx) : bool ≐
  let (p,px) = ppx, (q,qx) = qqx in
  s.i_step(p,q) ∧ px = qx
  ∨ s.x_en(p) ∧ ∃ox. x.trans(px, s.x_input(p), ox, qx)
                  ∧ q = s.x_output(p,ox)

m_ipfd_input(s, x)(qqx,i) : Q × Qx ≐
   let (q,qx) = qqx in s.input(q,i), qx

m_ipfd_wbehaved(s, x)(q,qx) : bool ≐
  s.x_en(q) ⇒ x.pre(qx, s.x_input(q)).
```

Here '`m_ipfd`' stands for 'map instantiating parameterised fine grain system keeping states distinct'. In the definition of `m_ipfd_step`, we see how `x_en` is used to identify when calls to the subsystem are enabled. In most sensible systems, we expect that it will never be possible to take an `i_step` when `x_en` is true, but we haven't found a need yet to specify this requirement. The function `x_input` is used to feed inputs to the subsystem, and function `x_output` processes the resulting outputs from the subsystem.

When a system is modelled as the parameterized fine grain system `s`, we assume that the granularity of `i_step`s is chosen sufficiently finely that, within the system behaviour modelled by a single `i_step`, there is no possibility for divergence or deadlock. Therefore, in defining `m_ipfd_wbehaved`, we need only consider that bad behaviour of `m_ipfd_step` can result if `x` is called when `x.pre` is false.

The map `mm_map` combines `m_ipfd` with the fine to medium grain map defined previously:

```
mm_map (s : Prm_fin_gs[Q,I,O,Ix,Ox])(x : Med_gs[Qx,Ix,Ox])
       : Med_gs[(Q × Qx),I,O]
    ≐  map_fm(m_ipfd(s, x)).
```

7 Refinement is Precongruence

Let `ps` be a parameterized fine grain system of type `Prm_fin_gs[Q,I,O,Ix,Ox]` with initial state `q`, let `sa` be an abstract medium grain subsystem of type `Prm_fin_gs[Qa,Ix,Ox]` with initial state `qa`, and let `sc` be a more concrete medium grain subsystem of type `Prm_fin_gs[Qc,Ix,Ox]` with initial state `qc`. The lemma `precong_lemma`:

```
⊢ refines_to(sa,sc)(qa,qc) ⇒
    refines_to(mm_map(ps)(sa), mm_map(ps)(sc))((q,qa),(q,qc))
```

states that the `refines_to` relation is a precongruence.

The proof is by coinduction, using the 'refines_to candidate':

```
rt_cand(sa,sc)(qqa,qqc) : bool ≐
  qqa.1 = qqc.1 and refines_to(sa,sc)(qqa.2,qqc.2)
```

to instantiate the coinduction lemma. Key foundational lemmas in the proof are

```
 ⊢ rt_cand(sa, sc)(qqa, qqc)                                    (4)
  ∧ at_progressive?(m_ipfd(ps,sa))(qqa)
  ∧ m_ipfd(ps, sc).step(qqc, rrc)
  ⇒
    ∃rra.  m_ipfd(ps, sa).step(qqa, rra)
            ∧ rt_cand(sa, sc)(rra, rrc)

⊢ rt_cand(sa,sc)(qqa,qqc)
  ∧ at_progressive?(m_ipfd(ps,sa))(qqa)
  ⇒
    at_progressive?(m_ipfd(ps,sc))(qqc).
```

Key intermediary lemmas are

```
⊢ rt_cand(sa,sc)(qqa,qqc)
  ∧ progressive?(m_ipfd(ps,sa))(qqa)
  ∧ ¬inf_chain(m_ipfd(ps,sa).step)(qqa)
  ⇒
    ¬inf_chain(m_ipfd(ps,sc).step)(qqc),
```

proven by coinduction on `inf_chain(m_ipfd(ps,sa).step)`, and

```
⊢ rt_cand(sa, sc)(qqa, qqc)
  ∧ progressive?(m_ipfd(ps,sa))(qqa)
  ∧ star(m_ipfd(ps, sc).step)(qqc, rrc)
  ⇒
    ∃rra. star(m_ipfd(ps, sa).step)(qqa, rra)
           ∧ rt_cand(sa, sc)(rra, rrc),
```

proven by induction on `star(m_ipfd(ps, sc).step)` using an inductive characterisation of `star(R)` with `R` steps successively added on the left, and use of lemma (4) above.

8 Example Specification

We give here an example of a specification of an ADT (abstract data type) of finite sets as a medium grain system, and an implementation as a fine grain system. We show the correctness statement for the implementation in terms of our *refines-to* relation.

8.1 Sets Specification

We consider the ADT to be parameterised by a type `T` of elements, to have operators:

```
bool empty ()     test if empty
void insert(T)    insert a possibly new element
void remove (T)   remove an existing element
T choose ()       choose an element
bool member (T)   test if an element is in the set
```

and to have a constructor `null` for the empty set.

We introduce datatypes for the input and outputs of both fine and medium grain systems.

```
IType [T:TYPE+] : DATATYPE
 BEGIN
   i_empty : i_empty?
   i_insert(i_insert_arg : T) : i_insert?
   i_remove(i_remove_arg : T) : i_remove?
   i_choose : i_choose?
   i_member(i_member_arg : T) : i_member?
 END IType

OType [T:TYPE+] : DATATYPE
 BEGIN
   o_empty(o_empty_val : bool) : o_empty?
   o_insert : o_insert?
   o_remove : o_remove?
   o_choose(o_choose_val : T) : o_choose?
   o_member(o_member_val : bool) : o_member?
 END OType
```

Such datatype statements in PVS declare constructors, recognisers, and field selectors, and introduce various auxiliary definitions and property axioms.

The medium grain system for sets is:

```
a_sys : Med_gs[AState,IType,OType] ≐
  ⟨ pre := a_pre, trans := a_trans ⟩,
```

where

```
AState : TYPE ≐ 𝒫(T)

a_trans(p,ip,op,q) : bool ≐
   cases ip of
     i_empty :     q = p ∧ op = o_empty(empty?(p)),
     i_insert(x) : q = add(x,p) ∧ op = o_insert,
     i_remove(x) : member(x,p) ∧
```

```
                q = remove(x,p) ∧ op = o_remove,
    i_choose :  nonempty?(p) ∧
                q = p ∧ op = o_choose(choose(p)),
    i_member(x) : q = p ∧ op = o_member(member(x,p))
  endcases

a_pre(p,ip) : bool ≐
  cases ip of
    i_empty :     true,
    i_insert(x) : true,
    i_remove(x) : member(x,p),
    i_choose :    nonempty?(p),
    i_member(x) : true
  endcases.
```

An initial state for the empty set is:

```
a_null : AState ≐ emptyset[T].
```

Here we have employed definitions such as `member`, `remove` and `emptyset` from PVS's standard sets-as-predicates library.

Note that we only specify that the `remove` operation be well behaved and terminate if it happens that that element we are trying to remove is indeed initially contained in the set.

We introduce a non-deterministic `choose` operation to pick some element from a set. It is defined in terms of the `choose` function on sets, which in turn makes use of the Hilbert epsilon operator in PVS's type theory. No requirement is placed on the behaviour of `choose` if the set is empty.

8.2 Sets Implementation

We base our implementation on lists. We will require these lists to not contain duplicates when we come to proving the correctness statement we show in the next subsection.

The type of states is:

```
CFState : TYPE ≐
  ⟨ pc : nat,
    sys_input : IType,
    set : list[T],
    tvar : T,
    tsvar : list[T],
    bvar : bool  ⟩
```

When executing an operation, we keep the input value to the fine grain system stored in the field `sys_input`. This field not only holds the input value (if any) of the operation, but also indicates *which* operation is currently executing. We use the predicate:

```
at_proc(p : pred[IType])(u) : bool ≐ p(u.sys_input)
```

to indicate which procedure we are currently in.

The field `pc` is the program counter, field `set` holds the list representation of the set, and fields `tvar`, `tsvar`, and `bvar` are temporary variables intended for use within operations.

The system definition is:

```
cf_sys : Fin_gs[CFState,IType,OType] ≐
 ⟨ run   := cf_run,
   input := cf_input,
   step  := cf_step,
   output := cf_output,
   wbehaved := λu. true  ⟩.
```

The system is in a run state when the `pc` is non-zero:

```
cf_run(u) : bool ≐ u.pc > 0.
```

Operations always start with a `pc` of `1`:

```
cf_input(u,ip) : CFState ≐ u with [sys_input := ip, pc := 1].
```

The step relation is composed from step relations for each operation:

```
cf_step(u,v) : bool ≐
   cases u.sys_input of
     i_empty     : cf_empty_step(u,v),
     i_insert(x) : cf_insert_step(u,v),
     i_remove(x) : cf_remove_step(u,v),
     i_choose    : cf_choose_step(u,v),
     i_member(x) : cf_member_step(u,v)
   endcases.
```

Examples of step relations for operations are:

```
cf_remove_step(u,v) : bool ≐
   cf_remove_step_1(u,v)
   ∨ cf_remove_step_2(u,v)
   ∨ cf_remove_step_3(u,v)
   ∨ cf_remove_step_4(u,v)
   ∨ cf_remove_step_5(u,v)

cf_remove_step_2(u,v) : bool ≐
   at_pc(2)(u) ∧ cons?(u.set) ∧ v = u with [pc := 3]

cf_remove_step_4(u,v) : bool ≐
   at_pc(3)(u)
   ∧ cons?(u.set)
```

```
    ∧ car(u.set) = i_remove_arg(u.sys_input)
    ∧ v = u with [pc := 2, set := cdr(u.set)]

  cf_choose_step(u,v) : bool ≐
    at_pc(1)(u)
    ∧ cons?(u.set)
    ∧ v = u with [pc := 0, tvar := car(u.set)]
```

Note how we implement `choose` by simply returning the head element of the list.

Selector functions on Pvs datatype are partial functions, only total on the relevant subtype of the datatype. For example, above, `remove_step_2` is the only way of reaching `at_pc(3)`, but to make `remove_step_4` type check in Pvs, we have to again check the `cons?`ness of `u.set`. We could regard the possibility of deadlock that this repeated check implies as a way of modelling the exception that might be raised in actual code if we reached step 4 and `u.set` were not a `cons`.

The output function is:

```
  cf_output(u) : OType ≐
    cases u.sys_input of
      i_empty :     o_empty(null?(u.set)),
      i_insert(x) : o_insert,
      i_remove(x) : o_remove,
      i_choose :    o_choose(u.tvar),
      i_member(x) : o_member(u.bvar)
    endcases.
```

The correctness of the output function relies on the preservation of the `sys_input` field from when an operation is started.

The initial state of the implementation is:

```
  cf_null : CFState ≐
    ⟨ pc := 0,
      sys_input := i_empty,
      set := null[T],
      tvar := ϵx. true,
      tsvar := null[T],
      bvar := false  ⟩.
```

The only important values here are for `pc` and `set`. The other values are just placeholders.

8.3 Correctness Statement

We construct a medium grain black-box abstraction of this system as follows:

```
CMState : TYPE ≐ CFState,
cm_sys : Med_gs[CMState,IType,OType] ≐ map_fm(cf_sys),
cm_null : CMState ≐ cf_null.
```

The theorem that our implementation is a correct implementation of the sets specification is then:

```
⊢ refines_to(a_sys, cm_sys)(a_null, cm_null).
```

9 Conclusions

We have introduced a refinement relation *refines-to* that can be used in correctness specifications for a very general class of programs. For example it is applicable to abstract data types, modules and classes in imperative and object oriented languages. The relation's merits include

- it captures total correctness requirements,
- it has a simple intuitive operational reading,
- it captures expectations about the covariant nature of outputs and the contravariant nature of inputs under refinement,
- it is a precongruence with respect to a general class of environments,
- standard proof obligations used for refinement in VDM can be derived from it.

While its use should make specifications significantly clearer than they might otherwise be, it doesn't make verification tasks any easier.

We are currently exploring the use of *refines-to* in specifying and verifying garbage collection algorithms. A refinement approach is appealing because it allows the use of black-box abstract data types for both the specification and implementation of garbage-collected heap memory. In previous work of ours in this area [8], we specified correctness using linear temporal logic assertions that had to refer to internal details of the garbage collection algorithm.

References

[1] Samson Abramsky. A note on reactive refinement. Personal Communication, May 19th 1999.
[2] Rajeev Alur and Thomas Henzinger. Reactive modules. *Formal methods in System Design*, 15:7–48, 1999.
[3] Rajeev Alur, Thomas A. Henzinger, Orna Kupfermann, and Moshe Y. Vardi. Alternating refinement relations. In *CONCUR '98*, LNCS. Springer Verlag, 1998.
[4] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
[5] David L. Dill. *Hierarchical Verification of Speed Independent Asynchonous Circuits*. MIT, 1988.
[6] Matthew Hennessey. *A theory of Testing*. MIT, 1989.

[7] Ulrich Hensel and Bart Jacobs. Coalgebraic theories of sequences in PVS. *Journal of Logic and Computation*, 9(4):463–500, 1999.

[8] Paul B. Jackson. Verifying a garbage collection algorithm. In Jim Grundy and Malcolm Newey, editors, *11th International Conference on Theorem Proving in Higher-Order Logics: TPHOLs'98*, volume 1479 of *Lecture Notes in Computer Science*, pages 225–244. Springer-Verlag, September 1998.

[9] Bart Jacobs. Behaviour-refinement of coalgebraic specifications with coinductive correctness proofs. In *Proceedings of TAPSOFT/FASE 1997*, LNCS. Springer Verlag, 1997.

[10] C. B. Jones. *Program Specification and Verification in VDM*. Prentice Hall, 2nd edition, 1990.

[11] R. Milner. Processes: a mathematical model of computing agents. In *Logic colloquium '73*, pages 157–173. North Holland, 1975.

[12] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[13] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992. See http://www.csl.sri.com/pvs.html for up-to-date information on PVS.