Parallel Refinement Mechanisms

Paul Z. Kolano

University of California, Santa Barbara

Abstract

Refinement is a fundamental design technique that has often challenged the "formal methods" community. In most cases, mathematical elegance and proof manageability have been chosen over flexibility and freedom, which are often needed in practice to deal with unexpected or critical situations. The issue of refinement becomes even more critical when dealing with real-time systems where time analysis is a crucial factor. In this case, the literature exhibits only a few, fairly limited proposals. In this paper, we propose general refinement mechanisms for real-time systems that allow several types of implementation strategies to be specified in a fairly natural way. Not surprisingly, generality has a price in terms of complexity. In our approach, however, this price is paid only when necessary. Furthermore, the proof system is amenable both for traditional hand-proofs, based on human ingenuity and only partially formalized, and for fully formalized, tool-supported proofs.

The following is an excerpt from [Kol 99]. It is assumed that the reader is already familiar with ASTRAL [CGK 97] and PVS [COR 95].

Chapter VII: Interlevel Refinement

Whether in programming languages or formal specification languages, refinement is the process of moving from an abstract design level to a concrete implementation by describing how the components in each upper level are implemented in the lower level. The left side of figure VII-1 shows the process of refinement, where each abstraction layer is depicted as a box. Each lower level box describes the implementation of the box above. Eventually, the complete system description is reached in the right side of the figure. Refinement allows designers to describe a system from the top down in more and more detail. That is, the desired behavior of each individual component is assumed and then the interactions between the components are specified. This allows designers to look at the components that make up the system and their interactions without looking at how each component is implemented. In this way, the design of a system can be modularized into different layers of abstraction. In formal methods, this allows the analysis of each abstraction layer to be proved without knowledge of how components in that layer are implemented. Each lower level component is then shown to implement the behavior that was assumed in

the upper level. This allows the analysis that was performed in the upper level to be preserved in the lower level. Proofs of properties in the upper level hold for all lower level implementations that meet the assumed behavior. It also simplifies the analysis of the upper level since the upper level does not need to be specified in as much detail as the lower level, so the state space is smaller in size than would be the case when reasoning about a complete implementation. This means that automated analysis techniques have a greater chance of success in the upper level. In real-time systems, refinement is more difficult that in untimed systems because not only do functional requirements need to be preserved, but also timing requirements. ASTRAL is well-suited for refinement since each process has a well-defined interface of what it relies on and what it guarantees, namely the environmental assumptions and imported variable clauses, and the invariants and schedules. This means that the lower level must preserve the invariants and schedules.

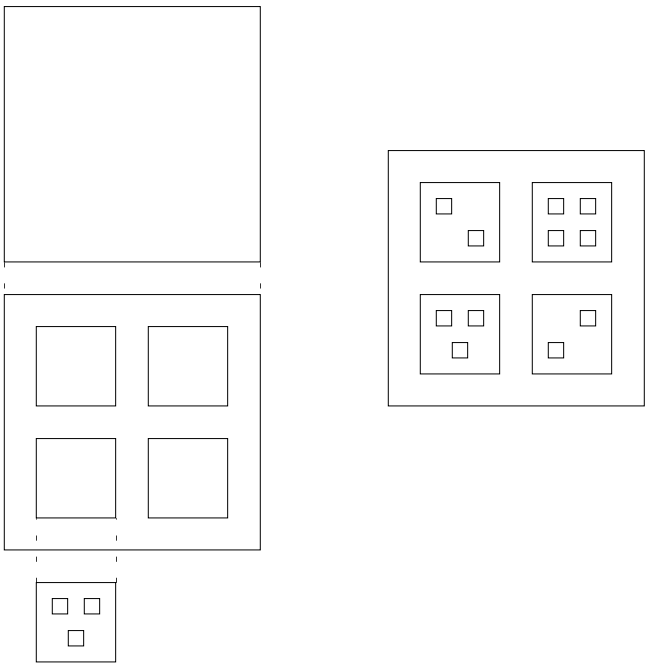

Figure VII-1: Refinement

VII.1. Sequential Refinement Mechanism

A refinement mechanism for ASTRAL was defined in [CKM 95]. In this definition, an ASTRAL process specification consists of a sequence of levels where the behavior of each level is implemented by the next lower level in the sequence. Given two ASTRAL process level specifications PU and PL, where PL is a refinement of PU, the implementation statement IMPL defines a mapping from all the types, constants, variables, and transitions of PU into their corresponding terms in PL. A type, constant, variable, or transition of PL representing the implementation of a corresponding term in PU is referred to as a mapped type, constant, variable, or transition. PL can also introduce types, constants and/or variables that are not

mapped. These are referred to as the new types, constants, or variables of PL. Note that PL cannot introduce any new transitions (i.e. each transition of PL must be a mapped transition). A transition of PU can be mapped into a sequence of transitions, a selection of transitions, or any combinations thereof.

A selection mapping of the form TU == A1 & TL.1 | A2 & TL.2 | ... | An & TL.n, is defined such that when the upper level transition TU fires, one and only one lower level transition TL.j fires, where TL.j can only fire when both its entry assertion and its associated "guard" Aj are true. The left side of figure VII.1.1 depicts a selection of transitions.

A sequence mapping of the form TU == WHEN EntryL DO TL.1 BEFORE TL.2 BEFORE ... BEFORE TL.n OD, defines a mapping such that the sequence of transition TL.1; ...; TL.n is enabled (i.e. can start) whenever EntryL evaluates to true. Once the sequence has started, it cannot be interrupted until all of its transitions have been executed in order. The starting time of the upper level transition TU corresponds to the starting time of the sequence (which is not necessarily equal to the starting time of TL.1 because of a possible delay between the time when the sequence starts and the time when TL.1 becomes enabled), while the ending time of TU corresponds to the ending time of the last transition in the sequence, TL.n. Note that the only transition that can modify the value of a mapped variable is the last transition in the sequence. This further constraint is a consequence of the ASTRAL communication model. That is, in the upper level, the new values of the variables affected by TU are broadcast when TU terminates. Thus, mapped variables of PL can be modified only when the sequence implementing TU ends. The right side of figure VII.1.1 depicts a sequence of transitions.

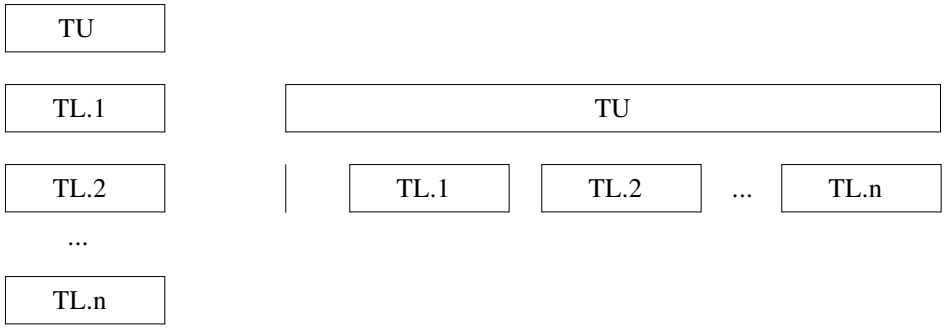

Figure VII.1.1: Selection and sequence mappings

VII.2. Proof Obligations for Sequential Refinement Mechanism

The interlevel proofs consist of showing that the mapping for each upper level transition is correctly implemented by the corresponding sequence, selection, or combination thereof in the next lower level. For selections, it must be shown that whenever the upper level transition TU fires, one of the lower level transitions TL.j fires, that the effect of each TL.j is equivalent to the effect of TU, and that the duration of each TL.j is equal to the duration of TU. These obligations are, respectively

(S0) IMPL(EntryU) ↔ A1 & EntryL.1 | ... | An & EntryL.n
(S1j) A.j′ & EntryL.j′ & ExitL.j → IMPL(ExitU)
(S2) DurL.1 = DurL.2 = ... = DurL.n = DurU

For sequences, it must be shown that the sequence is enabled if and only if TU is enabled, that the effect of the sequence is equivalent to the effect of TU, and that the duration of the sequence (including any initial delay after EntryL is true) is equal to the duration of TU. These are shown by the n+2 incremental proof obligations

(P0) IMPL(EntryU) ↔ EntryL
(Pj+1) past(EntryL, Now - DurU) & Start(TL.1, t1) & ... & Start(TL.j, tj)
→ EXISTS tj+1: Time
(tj+1 ≥ tj + DurL.j & tj+1 + $\sum_{k=j+1}^{n}$ DurL.k ≤ Now
& past(EntryL.j+1, tj+1)) where in obligation (n), "≤ Now" is replaced by "= Now"
(Pn+1) past(EntryL, Now - DurU) & past(ExitL.1, t1 + DurL.1) & ... & ExitL.n → IMPL(ExitU)

The idea of the selection and sequence obligations is that whenever an upper level transition is enabled, some lower level sequence or selection will be enabled because the entry assertions are equivalent. Similarly, whenever an upper level transition ends, some lower level sequence or selection will end because the durations are the same. Finally, whenever an upper level transition produces some effect, the lower level transition will produce an equivalent effect because the IMPL of the exit assertion of the upper level transition holds at the end of the lower level sequence or selection. This means that the upper and lower levels will have equivalent executions.

The SDE supports the implementation mechanism of [CKM 95]. Each level below the top level contains an implementation clause that describes the IMPL mapping between that level and the level above it. The validation component of the SDE checks the implementation clause for various errors such as not mapping an item in the upper level, not mapping a transition in the lower level, mapping an item more than once, etc. The VCG component of the SDE uses the information in the implementation clause to construct the above proof obligations. The expressions IMPL(EntryU) and IMPL(ExitU) used in the proof obligations are replaced by the appropriate lower level expressions as defined by the IMPL mapping. The VCG algorithm also includes corrections to two of the problems that have been found in [CKM 95]. The two problems that have been corrected are the proof obligations for arbitrary sequence and selection combinations and the further assumptions algorithm. These problems as well as additional problems are discussed in the next section.

VII.3. Problems with Sequential Refinement Mechanism

Several problems exist in the work of [CKM 95].

VII.3.1. Arbitrary Sequences and Selections

The exact proof obligations for simple sequences and selections are given in [CKM 95], but the proof obligations that must be generated for an arbitrary combination of sequences and selections are not

discussed. While implementing the VCG component of the SDE, these proof obligations were developed. For example, consider the mapping

```
TU == WHEN EntryL DO ( A0 & ( A1 & TL.1
                            | A2 & (TL.2 BEFORE TL.3))
                      | A4 & TL.4) BEFORE TL.5 OD.
```

This mapping consists of nested sequences and selections. For arbitrary combinations of sequences and selections, the proof obligations are generated by constructing a set of simple sequences, such that all possible sequences that can occur in the arbitrary mapping are represented. This is done by "distributing" the selection portions of the mapping until a selection of simple sequences is obtained. Since all the mappings in the set are sequences, the existing sequence proof obligations can then be generated and proven to show that the behavior of the arbitrary mapping is equivalent to that of the upper level transition. For the above mapping, the set of sequences is

```
WHEN EntryL DO TL.1' BEFORE TL.5 OD
WHEN EntryL DO TL.2' BEFORE TL.3 BEFORE TL.5 OD
WHEN EntryL DO TL.4' BEFORE TL.5 OD,
```

where entry(TL.1′) = A0 & A1 & entry(TL.1), entry(TL.2′) = A0 & A2 & entry(TL.2), entry(TL.4′) = A4 & entry(TL.4), and exit(TL.j′) = exit(TL.j).

VII.3.2. Soundness of Proof Obligations

The proof obligations presented in [CKM 95] for interlevel refinement are unsound. First, there is no obligation to prove that the lower level begins execution in a state that is consistent with the initial state of the upper level. This obligation is stated as "Initial_ll → IMPL(Initial_ul)". Without this obligation, the other obligations can hold and yet the refinement does not preserve the properties of the upper level. For example, consider the following specification fragments.

```
upper level                 lower level
INITIAL                     IMPL(x) == x
   x = 5                    IMPL(t) == t
INVARIANT                   INITIAL
   x ≥ 0                       x = -2
TRANSITION t                TRANSITION t
   ...                         ...
```

In the initial state, the lower level does not preserve the invariant of the upper level since $x < 0$, but the proof obligations hold, so this allows us to derive IMPL(Invariant_ul) = TRUE, which is not true, so the proof obligations are unsound.

The obligations for sequences are also unsound. Consider the obligation P1 for sequences as given in [CKM 95].

(P1) past(EntryL, t0)
→ EXISTS t1: Time (t1 ≥ t0 & t1 + $\sum_{k=j+1}^{n}$ DurL.k ≤ Now & past(EntryL.1, t1))

This obligation states that if EntryL held at t0 (i.e. the sequence started at t0), then the entry of the first transition in the sequence (i.e. TL.1) must hold such that there is enough time left to complete the sequence. This does not exclude the possibility, however, of another transition being enabled at t1 and nondeterministically firing, thereby delaying or disabling the execution of TL.1. Suppose the sequence is a refinement of the upper level transition tr_ul. This means that when tr_ul fires in the upper level, the sequence for tr_ul starts to fire in the lower level, but during execution, another sequence or selection begins to fire, corresponding to a transition in the upper level firing while tr_ul is firing, which is not possible by the trans_mutex axiom.

To correct the obligations, the clause "past(EntryL.n, ti)" is changed to "past(Start(TL.n, ti), ti)". The change for the P1 obligation is shown below.

(P1′) past(EntryL, t0)
→ EXISTS t1: Time (t1 ≥ t0 & t1 + $\sum_{k=j+1}^{n}$ DurL.k ≤ Now & past(Start(TL.1, t1), t1))

This forces the first transition of the sequence to occur after the sequence begins execution. The obligations P2 through Pn need to be changed similarly.

VII.3.3. Further Assumptions and Restrictions Algorithm

In ASTRAL specifications, it is possible to specify implementational restrictions using further assumptions clauses. In these clauses, the domains of constants can be limited using constant refinement clauses and nondeterministic choices between transitions can be restricted using transition selection clauses. To show that a lower level is consistent with an upper level, it is necessary to show that the implementational choices specified in the upper level further assumptions clauses have been implemented correctly in the lower levels. In [CKM 95], an algorithm is given to construct new entry assertions for transitions based on transition selection clauses. This algorithm contains an error and is shown in the left side of figure VII.3.3.

Since REntryUj is initialized to true, the expression "REntryUj | TMP" will always be true and thus "EntryUj & ~(REntryUj)" will always be false, meaning that no transition will be enabled. There is also a problem in initializing TMP to true when TUj is not in any OpSeti or is in every ROpSeti when it is in OpSeti. In these cases, TMP will be true, thus "REntryUj | TMP" will always be true and again "EntryUj & ~(REntryUj)" will always be false. The algorithm can be fixed as shown in the right side of figure VII.3.3.

```
Foreach T_Uj in P_U do
  REntry_Uj := true
  Foreach R_i in TS do
    TMP := true
    if T_Uj ∈ {OpSet_i} ∧ TU_j ∉ {ROpSet_i} then
      Foreach T_Ul in P_U do
        case T_Ul ∈ {OpSet_i} ∧ T_Ul ≠ T_Uj do
          TMP := TMP & Entry_Ul od
        case T_Ul ∉ {OpSet_i} do
          TMP := TMP & ~Entry_Ul od
      od
      TMP := TMP & Bool_i
    fi
    REntry_Uj := REntry_Uj | TMP
  od
  REntry_Uj := Entry_Uj & ~(REntry_Uj)
od
```

```
Foreach T_Uj in P_U do
  REntry_Uj := false
  Foreach R_i in TS do
    if T_Uj ∈ {OpSet_i} ∧ TU_j ∉ {ROpSet_i} then
      TMP := true
      Foreach T_Ul in P_U do
        case T_Ul ∈ {OpSet_i} ∧ T_Ul ≠ T_Uj do
          TMP := TMP & Entry_Ul od
        case T_Ul ∉ {OpSet_i} do
          TMP := TMP & ~Entry_Ul od
      od
      TMP := TMP & Bool_i
    else TMP := false
    fi
    REntry_Uj := REntry_Uj | TMP
  od
  REntry_Uj := Entry_Uj & ~(REntry_Uj)
od
```

Figure VII.3.3. Original and fixed entry assertion construction algorithms

VII.3.4. IMPL Mapping

The IMPL mappings for types, constants, and variables are not discussed in [CKM 95], but are assumed to be an extension of the mappings in [AK 85], modified to include ASTRAL constructs. The mappings in [AK 85], however, do not consider any nontrivial type mappings, thus do not allow the IMPL translation of an arbitrary expression to be constructed. For example, consider an upper level with a type S: set of T and a variable v_s: S. In the lower level, the specifier may wish to implement S and v_s as L: list of T and v_l: L, such that if an element of type T is in the set v_s, the element is somewhere on the list v_l. The IMPL mapping can be defined as IMPL(S) == L and IMPL(v_s) == v_l. Suppose an entry assertion in a transition of the upper level states that "t ISIN v_s", where t is an element of type T. The proof obligations require IMPL(Entry(tr_ul)) be constructed in order to attempt the proofs. There is no mention in [CKM 95] or [AK 85], however, of how to construct the lower level expression for such a type mapping. If only variables are transformed, the entry assertion becomes "IMPL(t) ISIN v_l", but v_l is a list and ISIN is an operator on sets. It is thus necessary to define IMPL mappings in much more detail to be able to attempt the proof obligations. A full discussion of the revised IMPL mapping is presented in section VII.4.3.

VII.3.5. Expressiveness

Besides errors and omissions in the mechanism itself, the refinement mechanism in [CKM 95] also suffers from a lack of expressiveness and flexibility. That is, for many systems, there are realistic and useful refinements that cannot be expressed using this refinement mechanism. Consider the system shown in figure VII.3.5-1.

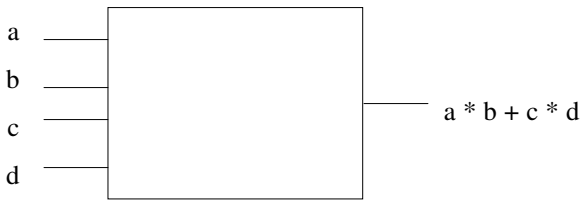

Figure VII.3.5-1: Mult_Add circuit

This system is a circuit that computes the value of a * b + c * d, given inputs a, b, c, and d. The ASTRAL specification for the circuit is shown below.

```
PROCESS SPECIFICATION     Mult_Add
    EXPORT
        compute, output
    CONSTANT
        dur1: pos_real
    VARIABLE
        output: integer
    INITIAL
        TRUE
    AXIOM
        TRUE
    INVARIANT
        FORALL t1: time, a, b, c, d: integer
            (   Start(compute(a, b, c, d), t1)
            &   Start(compute, t1)
        →       FORALL t2: time
                    (   t1 + dur1 ≤ t2
                    &   t2 ≤ now
                →       past(output, t2) = a * b + c * d))
    TRANSITION    compute(a, b, c, d: integer)
        ENTRY       [TIME:    dur1]
            TRUE
        EXIT
            output = a * b + c * d
END   Mult_Add
```

A reasonable refinement of the Mult_Add circuit is shown in figure VII.3.5-2.

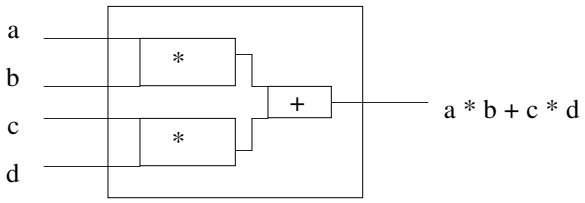

Figure VII.3.5-2: Refined Mult_Add circuit

Here, the refinement of the system consists of two multipliers, which compute a * b and c * d in parallel and then an adder that adds the products together and produces the sum. Although this refinement is a realistic refinement of the system, it cannot be expressed using the [CKM 95] refinement mechanism. This is because there is no notion of parallelism in that mechanism. In the end, the mechanism can only

refine a system into a choice between sequences of transitions. Thus, the closest execution that could be expressed is a nondeterministic choice between computing a * b first or c * d first as shown below, which does not capture the essential aspect of the desired refinement. Namely, it does not capture the parallelism between the two multipliers.

```
IMPL(compute(a, b, c, d)) ==
        WHEN TRUE DO  ( multiply(a, b) BEFORE multiply(c, d)
                      | multiply(c, d) BEFORE multiply(a, b))
                         BEFORE add(a * b, c * d) OD
```

The motivation for the development of a parallel refinement mechanism for ASTRAL is to support the expression of any reasonable refinement that a specifier may wish to design, such as the one for the Mult_Add system.

VII.4. Parallel Refinement Mechanism

In parallel refinement, an upper level transition may be implemented by a dynamic set of lower level transitions. To guarantee that an upper level transition is correctly implemented by the lower level, it is necessary to define what occurs in the lower level when a transition is executed in the upper level, so that it can be shown that it will only occur when an upper level transition fires and that the effect will be equivalent.

VII.4.1. Parallel Sequences and Selections

The first attempt at defining parallel transition mappings was to extend the sequence and selection mappings into parallel sequence and selection mappings. Thus, a "‖" operator could be allowed in transition mappings, such that "P1.tr1 ‖ P2.tr2" indicates that tr1 and tr2 occur in parallel on processes P1 and P2, respectively. With this addition, the compute transition of the Mult_Add circuit could be expressed as the following.

```
IMPL(compute(a, b, c, d)) ==
        WHEN TRUE DO  (    M1.multiply(a, b)
                      ||   M2.multiply(c, d)) BEFORE A1.add(a * b, c * d)
```

where M1 and M2 are the multipliers and A1 is the adder.

Although parallel sequences and selections work well for the example, they do not allow enough flexibility to express many reasonable refinements. For example, consider a production cell that executes a transition "produce" every time unit to indicate the production of an item. In a refinement of this system, the designer may wish to implement produce by defining two "staggered" production cells that each produce an item every two time units, thus effectively producing an item every time unit. The upper level production cell PU and the lower level production cells PL.1 and PL.2 are shown in figure VII.4.1. Note that the first transition executed on PU is an "initialize" transition that is used to represent the "warmup" time of the production cell in which no output is produced.

This refinement cannot be expressed using parallel sequences and selections because there is no sequence of parallel transitions at the lower level that corresponds directly to produce at the upper level. When produce starts in the upper level, one of the lower level produce's will start and when produce ends in the upper level, one of the lower level produce's will end and achieve the effect of upper level produce, but the produce that starts is not necessarily the produce that achieves the effect of the corresponding end.

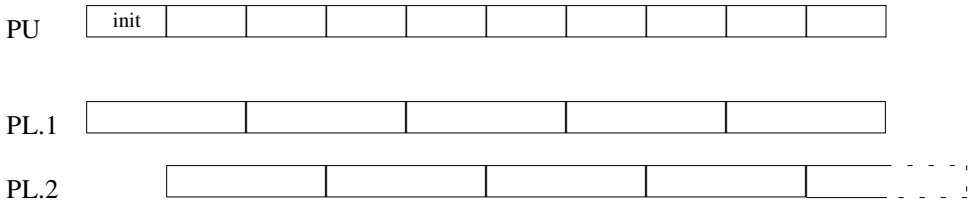

Figure VII.4.1: Production cell refinement

VII.4.2. Parallel Start and End Mappings

The actual transition mappings for the parallel refinement mechanism are based on the start, end, and call of each transition. For each upper level transition tr_ul, a start and end mapping must be defined as follows.

- IMPL(Start(tr_ul, now)) == wff_ll_start
- IMPL(End(tr_ul, now)) == wff_ll_end

If tr_ul is exported, a call mapping must also be defined.

- IMPL(Call(tr_ul, now)) == wff_ll_call

Here, wff_ll_start, wff_ll_end, and wff_ll_call are well-formed formulas using lower level transitions and variables. For the most part, the end and call mappings will always correspond to the end/call of some transition in the lower level, whereas the start mapping may correspond to the start of some transition or some combination of changes to variables/now/etc. Call mappings are restricted such that for every lower level exported transition tr_ll, Call(tr_ll) must be referenced in some upper level exported transition call mapping IMPL(Call(tr_ul, now)). This restriction expresses the fact that the interface of the process to the external environment cannot be changed. For parameterized transitions, only the call mapping may reference the parameters given to the transition. For an exported transition, the start and end parameters are the parameters generated at the calls to the transition. For non-exported transitions, the parameters are used to express nondeterminism so cannot be used in the mapping or else it would be necessary to prove that the upper and lower levels always make the same nondeterministic choices in the proof obligations.

With this mapping, the produce transition can be mapped as follows.

```
IMPL(Start(initialize, now)) ==
        now = 0
    &   P2.1.Start(produce, now)
IMPL(End(initialize, now)) ==
        now = 1
```

```
IMPL(Start(produce, now)) ==
        IF now mod 2 = 0
        THEN PL.1.Start(produce, now)
        ELSE PL.2.Start(produce, now)
        FI
IMPL(End(produce, now)) ==
        IF now mod 2 = 0
        THEN PL.1.End(produce, now)
        ELSE PL.2.End(produce, now)
        FI
```

It is possible to show that any sequence or selection mapping as defined in [CKM 95] can be described by the start and end mappings. For a selection TU == A1 & TL.1 | A2 & TL.2 | ... | An & TL.n, the start of the upper level transition TU occurs whenever one of transitions TL.i starts and its associated guard holds. This is described by the start mapping

```
IMPL(start(TU, now)) ==
        (   A1 & PL.start(TL.1, now)
        |   A2 & PL.start(TL.2, now)
        |   ...
        |   An & PL.start(TL.n, now))
```

The end of TU occurs whenever one of the transitions TL.i ends. This is described by the end mapping

```
IMPL(end(TU, now)) ==
        (   PL.end(TL.1, now)
        |   PL.end(TL.2, now)
        |   ...
        |   PL.end(TL.n, now))
```

For a sequence TU == WHEN EntryL DO TL.1 BEFORE TL.2 BEFORE ... BEFORE TL.n OD, the start of TU occurs whenever the entry condition of the sequence EntryL holds. This is described by the start mapping

```
IMPL(start(TU, now)) == EntryL
```

The end of TU occurs whenever the last transition TL.n ends. This is described by the end mapping

```
IMPL(end(TU, now)) == PL.end(TL.n, now)
```

Note that nothing is stated about the transitions that occur between when EntryL holds and TL.n ends. The rest of the sequence will appear in the proof obligations where it will need to be proven that each transition of the sequence occurs at the proper time to guarantee that the sequence takes Dur(TU) time and that the effect of TU is satisfied.

VII.4.3. Other Mappings

Besides transitions, the user must also define mappings for types, constants, and variables. For the most part, the constant and variable mappings are similar to the mappings used in [CKM 95]. In [CKM 95], however, the lower level only consisted of a single process type, so variable mappings only referred to the variables of a single process. In the parallel mechanism, variable mappings can refer to variables and transitions of any process in the refinement. For example, the mapping for a variable x can be defined

IMPL(x) == (P1.y + P2.z) / 2. That is, x is the average of the y variable of process P1 and the z variable of process P2. An additional consequence of having multiple lower level processes is that information must be provided about the processes that make up the lower level. Thus, the implementation section contains a processes clause similar to the clause in the global specification that describes all the process instances of the new lower level. The upper level process is the parallel composition of these process instances.

As discussed in section VII.3.4, the description of the type mappings in [CKM 95] and [AK 85] are not sufficient to construct the IMPL translation of an arbitrary expression. The example used in section VII.3.4 was a set type being mapped to a list type. This created a problem because the set operators are not valid on lists. In general, there is a problem any time an upper level type T is mapped to a lower level type IMPL(T) that is not "compatible" with T. We define types T and IMPL(T) to be compatible iff:

- T is an undefined type
- T is identical to IMPL(T)
- T is a list of E and IMPL(T) is a list of IE and E is compatible with IE
- T is a set of E and IMPL(T) is a set of IE and E is compatible with IE
- T is a structure of ID1: E1, ..., IDn: En and IMPL(T) is a structure of ID1: IE1, ..., IDn: IEn and Ei is compatible with IEi
- IMPL(T) is a typedef of E and T is compatible with E

Note that type mappings are restricted such that builtin types cannot be mapped and that any alias or subtype of a given supertype can only be mapped if no other alias or subtype has been mapped. For example, the types T1 and T2: typedef t1: T1 (P(t1)) cannot both be mapped. In this restriction, the builtin types integer and time are assumed to be subtypes of the supertype real.

Examples of compatible types are:

(1) T: (e1, e2)
IMPL(T): (e1, e2)
(2) T: list of real
IMPL(T): list of integer

Examples of incompatible types are:

(1) T: (open, closed)
IMPL(T): (open, closed, opening, closing)
(2) T: list of integer
IMPL(T): set of integer
(3) T: list of bool
IMPL(T): integer
(4) T: structure of (i1: integer, i2: integer)
IMPL(T): structure of (j1: integer, j2: integer)

A more complete definition of the IMPL mapping is given below. The IMPL mapping describes how items in an upper level are implemented by items in a lower level. The items of the upper level include variables, constants, types, and transitions. In addition, the IMPL mapping must describe how upper level expressions are transformed into lower level expressions. In many cases, namely when variables and

constants are mapped to expressions of compatible types, the basic mappings are sufficient to transform upper level expressions into lower level expressions. When mappings occur between incompatible types, however, the basic mappings must be supplemented with additional mapping information.

For each upper level type T that is mapped to an incompatible lower level type IMPL(T) and for each variable or constant of type T that is mapped to a lower level expression of an incompatible type TL, a mapping must be defined for each operator op in the upper level that is used on an item of type T. For simplicity, assume that all operators are in prefix notation.

IMPL(op(v1: T1, ..., vi: T, ..., vn: Tn)) == f(IMPL(v1), ..., IMPL(vi), ..., IMPL(vn))

The operator mappings are restricted such that none of the timed operators (i.e. start, end, call, change, and past) can be mapped. The start, end, and call operators will always be mapped as a simple replacement mapping as described earlier. The past and change operators will always use the "natural" operator mapping. The natural mapping is defined as follows.

IMPL_0(op(v1: T1, ..., vn: Tn)) == op(IMPL(v1), ..., IMPL(vn))

In other words, the natural mapping for operators passes the IMPL construct through to its operands. For example, IMPL(past(A, t)) == past(IMPL(A), IMPL(t)) and IMPL(change(A, t)) = change(IMPL(A), IMPL(t)). The implementation of any operator that does not have an explicit mapping for its operand types is defined to be the natural operator mapping.

As an example of an operator mapping, consider the mapping from type S: set of T to L: list of T, where the element type T of S and L is integer. Suppose an expression "{1, 2, 3} SUBSET v_s" occurs in the upper level. S is not compatible with L, so the SUBSET operator must be mapped.

```
IMPL(SUBSET(s1: S, s2: S)) ==
        list_len(IMPL(s1)) ≠ list_len(IMPL(s2))
    &   FORALL i: integer
            (   1 ≤ i
            &   i ≤ list_len(IMPL(s1))
        →       EXISTS j: integer
                    (   1 ≤ j
                    &   j ≤ list_len(IMPL(s2))
                    &   IMPL(s2)[j] = IMPL(s1)[i]))
```

In this case, the implementation of subset is defined such that whenever s1 is a proper subset of s2 in the upper level, the lists corresponding to s1 and s2 in the lower level do not have the same length and every element that is on the list IMPL(s1) is on the list IMPL(s2).

There are several things to note about this mapping. First, IMPL is allowed to be recursive on the structure of the parse tree. That is, for an operator op(p1, ..., pn), IMPL(op(...)) is allowed to reference IMPL(p1), ..., IMPL(pn). This allows the operator mappings to be significantly simplified because it is not necessary to describe how each operand is mapped. The operand mappings are described individually in their own mappings that can be reused in each operator mapping. For example, in the mapping for

ISIN, it is not necessary to describe how a set of type S is translated into a list of type L. This is described by a separate mapping. As a consequence of allowing recursion, the translation of an upper level expression cannot simply traverse the parse tree of the expression and replace each mapped object by its right hand side. Instead, the replacement algorithm is directed by the IMPL mapping. That is, the replacement algorithm must call itself whenever IMPL is used in the right hand side of a mapping that is currently being used for replacement.

The other thing to note is that operators may take items other than variables as operands. When a variable v is given as an operand, IMPL(v) is well-defined since all variables must be mapped. The operators may also take explicitly-valued constants (e.g. 5, {3, 6}, etc.) and imported variables as operands. This means that an IMPL mapping must be defined to map these types of operands to an equivalent lower level expression of the correct type. In the above example, IMPL(s1) is referenced in the definition of SUBSET and the set {1, 2, 3} is used as an operand to SUBSET in the upper level, so IMPL must define how the set {1, 2, 3} is mapped to LISTDEF (1, 2, 3) and in general how any constant or imported set of integers is mapped to a list of integers. Like operators, a natural constant mapping is defined as follows.

IMPL_0(c: T) == c for any type T that has a builtin supertype

List and set constants are mapped using the natural operator mapping.

```
IMPL_0(LISTDEF(e1, ..., en)) ==
        LISTDEF(IMPL(e1), ..., IMPL(en))
IMPL_0({e1, ..., en}) ==
        {IMPL(e1), ..., IMPL(en)}
IMPL_0(SETDEF e: T (P(e))) ==
        SETDEF e: IMPL(T) (IMPL(P(e)))
```

In these mappings, the values of a builtin type are mapped to the same values, a list of elements is mapped to a list of the implementation of each element, and a set of elements is mapped to a set of the implementation of each element. For each operator mapping IMPL(op(p1, ..., pi: T, ..., pn)) that references IMPL(pi) such that IMPL(c: T) has not been defined and IMPL_0(pi) is either undefined or causes a type mismatch when exchanged for IMPL(pi), the user must define a mapping IMPL(c: T). If no such mapping is required, IMPL(c: T) is defined to be IMPL_0(c: T).

In general, an element of type T in the upper level may be mapped to more than one value of type IMPL(T) at the lower level. For example, consider the mapping from type S to type L. In this mapping, a set v_s in the upper level maps to a list v_l in the lower level such that v_l contains exactly the elements that are in v_s. Lists, however, are ordered, so the elements in v_s may occur in v_l in any order. Therefore, v_s maps to set_size(v_s) factorial different lists in the lower level. In general, it is undesirable to limit the value that can be chosen in the lower level, which in turn would limit implementational possibilities. For example, if type T was totally ordered, v_l could be chosen such that if t1 and t2 were in v_s and t1 < t2, then t1 would occur before t2 in v_l. In some cases, however, it is not possible to choose

one particular value at the lower level. If T is an undefined type, there is no way to describe a transformation from v_s to a specific v_l in the ASTRAL base logic because nothing is known about elements of type T.

To facilitate such mappings, the choose operator "choose e: T (P(e))" is introduced into the ASTRAL language, which corresponds to Hilbert's epsilon operator [??] epsilon(...). The value of the expression "choose e: T (P(e))" is an element e of type T, such that the ASTRAL predicate P(e) holds if such an element exists. If more than one such element exists, the operator nondeterministically chooses one of those elements. If no such element exists, the operator nondeterministically chooses some element of T.

With the choose operator in the language, defining element transformations becomes much simpler. For example, consider the mapping from elements of type S to elements of type L.

```
IMPL(v_s: S) ==
        choose v_l: L
            (   list_len(v_l) = set_size(IMPL_0(v_s))
            &   FORALL e: IMPL(T)
                    (   e ISIN IMPL_0(v_s)
                ↔       EXISTS i: integer
                            (   1 ≤ i
                            &   i ≤ list_len(v_l)
                            &   v_l[i] = e)))
```

This mapping states that a constant or imported variable v_s of type S is mapped to a list v_l of type L such that the length of v_l is equal to the cardinality of v_s and every element in v_s is on v_l. Note that in this mapping, the natural mapping IMPL_0 is referenced to avoid any reference to an upper level term (in this case v_s) in the right hand side of the mapping. Although it is possible to avoid referencing upper level terms in most cases, it is impossible to avoid this in all cases. In particular, when mapping constants, it is sometimes necessary to choose a replacement expression based on the actual value of the constant in the upper level. Most notably, when mapping enumerated types, it is necessary to reference upper level enumerated constants in the right hand side of the IMPL mapping. For example, consider an upper level enumerated type "gate_u: (open, closed)" that is mapped to a lower level type "gate_l: (open, closed, opening, closing)" such that closed maps to closed and open maps to one of open, opening, or closing. In this mapping, there is no way to map an arbitrary constant of type gate_u to a constant of type gate_l without selecting a value based on gate_u. To accommodate such mappings, a single case split is allowed on the upper level constant that is being mapped such that each case corresponds to an explicit constant value. For example, arbitrary gate_u constants can be mapped as follows:

```
IMPL(c: gate_u) ==
    CASE c OF
        open:
            choose e: gate_l
                (   e = open
                |   e = opening
                |   e = closing)
        close:
            closed
    ESAC
```

When c is an actual constant value, the IMPL replacement algorithm uses the case information to choose the correct replacement. When c is an imported variable, the right side of the mapping is substituted as is, which is well-defined since the upper level type must be globally defined and the interface to the process does not change from the top level so types available at the top level are still available at the lower levels.

VII.5. The Mult_Add Circuit

The specification of the refinement of the Mult_Add circuit in figure VII.3.5-2 is shown below using the new parallel refinement mechanism. Each multiplier has a single exported transition "multiply" that computes the product of two inputs. The adder has a single transition "add" that computes the sum of the outputs of the two multipliers.

```
PROCESS SPECIFICATION    Multiplier
  EXPORT
      multiply, product
  VARIABLE
      product: integer
  INITIAL
      TRUE
  TRANSITION   multiply(a, b: integer)
    ENTRY      [TIME:   2]
        EXISTS t: time
          (   end(multiply, t))
        →     now - end(multiply) ≥ 1
    EXIT
        product = a * b
END Mult
```

```
PROCESS SPECIFICATION Adder
  IMPORT
      M1, M2, M1.product, M2.product,
      M1.multiply, M2.multiply
  EXPORT
      sum
  VARIABLE
      sum: integer
  INITIAL
      TRUE
  TRANSITION   add
    ENTRY      [TIME:   1]
        M1.end(multiply, now)
      &  M2.end(multiply, now)
    EXIT
        sum = M1.product + M2.product
END  Add
```

The lower level consists of two instances of the Multiplier process type and one instance of the Adder process type.

```
PROCESSES
    M1, M2: Multiplier
    A1: Adder
```

The output variable of the upper process is mapped to the sum variable of the adder.

```
IMPL(output) == A1.sum
```

The duration of the compute transition is the sum of the multiply transition and the add transition in the lower level.

```
IMPL(dur1) == 3
```

When compute starts in the upper level on inputs a, b, c, and d, then multiply starts on M1 on inputs a and b, and multiply starts on M2 on inputs c and d. When compute ends in the upper level, add ends on A1. When compute is called in the upper level with inputs a, b, c, and d, multiply is called on M1 with inputs a and b and multiply is called on M2 with inputs c and d.

```
IMPL(start(compute, now)) ==
        M1.start(multiply, now)
    &   M2.start(multiply, now)
IMPL(end(compute, now)) == A1.end(add, now)
IMPL(call(compute(a, b, c, d), now)) ==
        M1.call(multiply(a, b), now)
    &   M2.call(multiply(c, d), now)
```

VII.6. Proof Obligations for Parallel Refinement Mechanism

The goal of the refinement proof obligations is to show that any properties that hold in the upper level hold in the lower level without actually reproving the upper level properties in the lower level. In order to show this, it must be shown that the lower level correctly implements the upper level. ASTRAL properties are interpreted over execution histories, which are described by the values of state variables and the start, end, and call times of transitions at all times in the past back to the initialization of the system. A lower level correctly implements an upper level if the implementation of the execution history of the upper level is equivalent to the execution history of the lower level. This corresponds to proving the following four statements.

(V) Any time a variable has one of a set S of possible values in the upper level, the implementation of the variable has one of a subset of the implementation of S in the lower level.

(C) Any time the implementation of a variable changes in the lower level, a transition ends in the upper level.

(S) Any time a transition starts in the upper level, the implementation of the transition starts in the lower level and vice-versa.

(E) Any time a transition ends in the upper level, the implementation of the transition ends in the lower level and vice-versa.

If these four items can be shown, then any property that holds in the upper level is preserved in the lower level because the structures over which the properties are interpreted is identical over the implementation mapping.

VII.6.1. Direct Proof Obligations

As a first attempt at defining the proof obligations for the parallel refinement mechanism, the proofs of (V), (C), (S), and (E) can be carried out directly. That is, they can each be expressed explicitly using the variable, start, and end mappings and then proved at all times. The equivalence can be proved inductively on the time domain. For simplicity, a discrete time domain will be used in the following discussion. Thus, in the proof obligations, all the mappings can be assumed to hold up until but not including some time T0, and then it must be proved that each mapping holds at T0.

Let UL_state of type [ul_bool_expr → bool] and LL_state of type [ll_bool_expr → bool] be uninterpreted functions such that UL_state(ul_bool_expr) is true iff ul_bool_expr holds in the upper level and LL_state(ll_bool_expr) is true iff ll_bool_expr holds in the lower level, where ul_bool_expr is a boolean expression involving upper level terms and ll_bool_expr is a boolean expression involving lower level terms. For example, the expression UL_state(start(compute, t)(t)) is true iff in the execution of the upper level process Mult_Add, compute starts at time t.

Two assumptions, A_const and A_call, will be made in the proof obligations. Let const_ul denote the set of constants in the upper level, cval_ul denote the set of possible constant values in the upper level, trans_ul denote the set of transitions in the upper level, and var_ul denote the set of variables in the upper level. A_const states that if a constant c_ul has one of a possible set of values s_cv_ul, then the implementation of c_ul has one of a subset of the values in s_cv_ul in the lower level.

```
(A_const)   FORALL c_ul: const_ul, s_cv_ul, s2_cv_ul: set(type(c_ul))
               (   s2_cv_ul CONTAINED_IN s_cv_ul
            →  (   UL_state(c_ul ISIN s_cv_ul)
               ↔ LL_state(IMPL(c_ul ISIN s2_cv_ul))))
```

A_call states that any time a transition tr_ul is called at the upper level, the implementation of a call to tr_ul occurs at the lower level.

```
(A_call)    FORALL t: time, tr_ul: trans_ul
               (   UL_state(call(tr_ul, t)(t))
               ↔ LL_state(IMPL(call(tr_ul, t))(t)))
```

The base case obligations are shown below. For variables, if v_ul has one of a possible set of values in s_cv_ul at time 0 in the upper level, then the implementation of v_ul has one of a subset of the values in s_cv_ul at time 0 in the lower level. For variable changes, no base case obligation is needed because no expression can change at time 0 by the definition of the change operator. For transitions, if tr_ul starts at time 0 in the upper level, then the implementation of the start of tr_ul must hold at time 0 in the lower level. Note that although tr_ul cannot end at time 0 in the upper level, the implementation of the end of tr_ul can be an arbitrary expression that does not necessarily involve transition ends, so can hold at time 0, thus the base case for ends must be similarly shown.

```
(V_b)    A_call & A_const
      → FORALL v_ul: var_ul, s_cv_ul, s2_cv_ul: set(type(v_ul))
           (   s2_cv_ul CONTAINED_IN s_cv_ul
         →     (   UL_state((v_ul ISIN s_cv_ul)(0))
               ↔ LL_state(IMPL(v_ul ISIN s2_cv_ul)(0))))

(S_b)    A_call & A_const
      → FORALL tr_ul: trans_ul
           (   UL_state(start(tr_ul, 0)(0))
           ↔ LL_state(IMPL(start(tr_ul, 0))(0)))

(E_b)    A_call & A_const
      → FORALL tr_ul: trans_ul
           (   UL_state(end(tr_ul, 0)(0))
           ↔ LL_state(IMPL(end(tr_ul, 0))(0)))
```

The inductive assumption A_induct is defined as follows.

```
(A_induct)    FORALL t: time
                 (   t < T0
              →     (   FORALL v_ul: var_ul, s_cv_ul, s2_cv_ul: set(type(v_ul))
                           (   s2_cv_ul CONTAINED_IN s_cv_ul
                         →     (   UL_state((v_ul ISIN s_cv_ul)(t))
                               ↔ LL_state(IMPL(v_ul ISIN s2_cv_ul)(t))))
                    &  FORALL tr_ul: trans_ul
                           (   UL_state(start(tr_ul, t)(t))
                           ↔ LL_state(IMPL(start(tr_ul, t))(t)))
                    &  FORALL tr_ul: trans_ul
                           (   UL_state(end(tr_ul, t)(t))
                           ↔ LL_state(IMPL(end(tr_ul, t))(t)))))
```

The induction step obligations are similar to the base case obligations, but they must be shown at an arbitrary time T0 instead of at time 0. In addition, the mappings are assumed up until T0. The induction step obligations are shown below.

```
(V_i)    A_call & A_const & A_induct
      → FORALL v_ul: var_ul, s_cv_ul, s2_cv_ul: set(type(v_ul))
           (   s2_cv_ul CONTAINED_IN s_cv_ul
         →     (   UL_state((v_ul ISIN s_cv_ul)(T0))
               ↔ LL_state(IMPL(v_ul ISIN s2_cv_ul)(T0))))

(C_i)    A_call & A_const & A_induct
      → FORALL v_ul: var_ul, tr_ul: trans_ul
           (   LL_state(change(IMPL(v_ul), T0)(T0))
         →     UL_state(end(tr_ul, T0)(T0)))

(S_i)    A_call & A_const & A_induct
      → FORALL tr_ul: trans_ul
           (   UL_state(start(tr_ul, T0)(T0))
           ↔ LL_state(IMPL(start(tr_ul, T0))(T0)))

(E_i)    A_call & A_const & A_induct
      → FORALL tr_ul: trans_ul
           (   UL_state(end(tr_ul, T0)(T0))
           ↔ LL_state(IMPL(end(tr_ul, T0))(T0)))
```

To prove these obligations, axioms must be introduced for the manipulation of expressions involving UL_state and LL_state. In general, new information can be deduced from any UL_state/LL_state expression by using the axiom system utilized for the intralevel proofs in PVS. For example, if UL_state(Start(tr1, t1)(t)) holds, it can be deduced that UL_state(Fired(tr1, t1)) holds and hence that UL_state(Entry(tr1, t1)) holds. In addition, axioms are defined for splitting and combining UL_state/LL_state expressions. For example, if LL_state(A & B) holds, it can be deduced that LL_state(A) & LL_state(B) hold, and vice-versa. There are also axioms to reduce UL_state/LL_state expressions to constants, such as UL_state(FALSE) = FALSE. Finally, there are axioms to deduce that information about the operating environment in the upper and lower levels are the same at all times. For example, for an imported variable P.v, if UL_state(P.v(t)), then we can deduce LL_state(P.v(t)). Basically, it can be assumed that all other processes besides the one being refined behave the same in the executions of the upper and lower levels.

These obligations are almost direct translations of the requirements (V), (C), (S), and (E) above. Therefore, they are sufficient to show that a lower level correctly implements an upper level. They suffer, however, from a number of major drawbacks. First, they require the introduction of a new axiom system to manipulate UL_state and LL_state expressions. This means that the user must learn an additional axiom system to perform the proofs. Additionally, they require the user to reason about both the upper and lower levels in the proofs. Finally, they require additional mechanisms to handle nondeterministic systems as discussed below.

In a specification with nondeterministic behavior such as multiple transitions enabled at the same time or a nondeterministic choice of variable values in an exit assertion, there is a need for some type of “oracle” that relates nondeterministic choices made in the upper and lower levels. That is, if a choice c is made in the upper level, then c is made in the lower level. Normally, the inductive hypothesis subsumes the need for an oracle, but it is necessary at the time the mapping is to be proved (i.e. time 0 in the base case and time T0 in the induction step). For instance, suppose the mapping for Start(tr1, t) is being proved, but whenever tr1 is enabled, tr2 is also enabled. Without an oracle stating that tr1 is chosen at t, there is no way to prove that if tr1 starts in the upper level that the implementation of tr1 will actually start in the lower level because the implementation of tr2 might actually start.

Such an oracle is difficult to define, given that mappings between an upper level and lower level can be complex expressions and the fact that while proving, it is not known if the lower level is actually a refinement of the upper level so that the same things will be enabled, same variable choices will occur, etc. A first attempt at defining an oracle might be:

- For transitions, if at time t, tr1, ..., trn are enabled in the upper level, the process is idle, and tri fires, then if at time t, IMPL(tr1), ..., IMPL(trn) are enabled in the lower level and the appropriate processes are idle, then IMPL(tri) occurs.
- For variables, if at time t, there exists a choice between values v1, ..., vn to give a variable var in an exit assertion in the upper level and vi is chosen, then if at time t, there exists a choice between values IMPL(v1), ..., IMPL(vn) to give an expression IMPL(var) in an exit assertion in the lower level, then IMPL(vi) is chosen.

Even if such an oracle could be defined, it would overly complicate the proofs of any nondeterministic system. In the end, proving (V), (C), (S), and (E) directly requires too much overhead for even simple implementations. Instead of proving these requirements directly, they will be proven indirectly in the actual proof obligations for the parallel refinement mechanism discussed in the next section.

VII.6.2. Indirect Proof Obligations

Instead of proving directly that the mappings hold at all times, it will be shown that the mappings hold indirectly by proving that they preserve the axiomatization of the ASTRAL abstract machine, thus preserve any reasoning performed in the upper level. This can be done by proving the implementation of each abstract machine axiom.

To perform the proofs, the following assumption must be made about calls to transitions in each lower level process.

```
impl_call: ASSUMPTION
    (FORALL (tr_ll, t1):
        Exported(tr_ll) AND
        Call(tr_ll, t1)(t1) IMPLIES
            (EXISTS (tr_ul):
                (IMPL(Call(tr_ul, t1)(t1)) IMPLIES
                    Call(tr_ll, t1)(t1)) AND
                IMPL(Call(tr_ul, t1)(t1))))
```

This assumption states that any time a lower level exported transition is called, there is some call mapping that references a call to the transition that holds at the same time. This means that if one transition of a "conjunctive" mapping is called, then all transitions of the mapping are called. That is, it is not possible for a lower level transition to be called such that the call mapping for some upper level transition does not hold. For example, consider the mapping for the compute transition of the Mult_Add circuit.

```
IMPL(call(compute(a, b, c, d), t)) ==
        M1.call(multiply(a, b), t)
    &   M2.call(multiply(c, d), t)
```

In this case, impl_call states that any time multiply is called on M1, multiply is called on M2 at the same time and vice-versa. Note that impl_call is not an environmental assumption. The call mappings explicitly state how calls at the lower level are generated from calls at the upper level. The impl_call

assumption expresses this fact and makes sure that lower level calls do not occur in any other way but as stated.

In the axiomatization of the ASTRAL abstract machine, the predicate "Fired(tr1, t1)" is used to denote that the transition tr1 fired at time t1. If Fired(tr1, t1) holds, then it is derivable that Start(tr1, t1)(t1) and End(tr1, t1 + Duration(tr1))(t1 + Duration(tr1)). Additionally, since End(tr1, t1)(t1) can only be derived when Fired(tr1, t1 - Duration(tr1)) holds and the time parameter of Fired is restricted to be nonnegative, it is known that an end can only occur at times greater than or equal to the duration of the transition. In the parallel refinement mechanism, the start and end of upper level transitions are mapped by the user, so it is unknown whether these properties of end still hold. Since the axioms rely on these properties, they must be proved explicitly as proof obligations. The impl_end1 obligation ensures that the mapped end of a transition can only occur after the mapped duration of the transition has elapsed.

```
impl_end1: OBLIGATION
    (FORALL (tr1, t1):
        IMPL(End(tr1, t1)(t1)) IMPLIES
            t1 ≥ IMPL(Duration(tr1)))
```

The impl_end2 obligation ensures that for every mapped start of a transition, there is a corresponding mapped end of the transition, that for every mapped end, there is a corresponding mapped start, and that mapped starts and mapped ends are separated by the mapped duration of the transition.

```
impl_end2: OBLIGATION
    (FORALL (tr1, t1, t2):
        t1 = t2 - IMPL(Duration(tr1)) IMPLIES
            (IMPL(Start(tr1, t1)(t1)) IFF
                IMPL(End(tr1, t2)(t2))))
```

The following obligations are the mappings of the ASTRAL abstract machine axioms. The impl_trans_entry obligation ensures that any time the mapped start of a transition occurs, the mapped entry assertion of the transition holds.

```
impl_trans_entry: OBLIGATION
    (FORALL (tr1, t1):
        IMPL(Start(tr1, t1)(t1)) IMPLIES
            IMPL(Entry(tr1, t1)))
```

The impl_trans_exit obligation ensures that any time the mapped end of a transition occurs, the mapped exit assertion of the transition holds.

```
impl_trans_exit: OBLIGATION
    (FORALL (tr1, t1):
        IMPL(End(tr1, t1)(t1)) IMPLIES
            IMPL(Exit(tr1, t1)))
```

The impl_trans_called obligation ensures that any time the mapped start of an exported transition occurs, a mapped call has been issued to the transition but not yet serviced.

```
impl_trans_called: OBLIGATION
    (FORALL (tr1, t1):
        IMPL(Start(tr1, t1)(t1)) AND Exported(Base_Trans(tr1)) IMPLIES
            IMPL(Issued_Call(Base_Trans(tr1), t1)))
```

The impl_trans_mutex obligation ensures that any time the mapped start of a transition occurs, no other mapped start of a transition can occur until the mapped duration of the transition has elapsed.

```
impl_trans_mutex: OBLIGATION
    (FORALL (tr1, t1):
        IMPL(Start(tr1, t1)(t1)) IMPLIES
            (FORALL (tr2):
                tr2 ≠ tr1 IMPLIES
                    NOT IMPL(Start(tr2, t1)(t1))) AND
            (FORALL (tr2, t2):
                t1 < t2 AND t2 < t1 + IMPL(Duration(tr1)) IMPLIES
                    NOT IMPL(Start(tr2, t2)(t2))))
```

The impl_trans_fire obligation ensures that any time the mapped entry assertion of a transition holds, a mapped call has been issued to the transition but not yet serviced if the transition is exported, and no mapped start of a transition has occurred within its mapped duration of the given time, a mapped start will occur.

```
impl_trans_fire: OBLIGATION
    (FORALL (t1):
        (EXISTS (tr1):
            IMPL(Enabled(tr1, t1))) AND
        (FORALL (tr2, t2):
            t1 - IMPL(Duration(tr2)) < t2 AND t2 < t1 IMPLIES
                NOT IMPL(Start(tr2, t2)(t2))) IMPLIES
        (EXISTS (tr1): IMPL(Start(tr1, t1)(t1))))
```

The impl_vars_no_change obligation ensures that mapped variables only change value when the mapped end of a transition occurs.

```
impl_vars_no_change: OBLIGATION
    (FORALL (t1, t3):
        t1 ≤ t3 AND
        (FORALL (tr2, t2):
            t1 < t2 AND t2 ≤ t3 IMPLIES
                NOT IMPL(End(tr2, t2)(t2))) IMPLIES
                    (FORALL (t2):
                        t1 ≤ t2 AND t2 ≤ t3 IMPLIES
                            IMPL(Vars_No_Change(t1, t2))))
```

The impl_initial_state obligation ensures that the mapped initial clause holds at time 0.

```
impl_initial_state: OBLIGATION
    IMPL(Initial(0))
```

Besides the abstract machine axioms, the local proofs of ASTRAL process specification can also reference the local axiom clause of the process. Since this clause can be used in proofs and the constants referenced in the clause can be implemented at the lower level, the mapping of the local axiom clause of the upper level must be proved as a proof obligation. The impl_local_axiom obligation ensures that the mapped

axiom clause holds at all times. In order to prove this obligation, it may be necessary to specify local axioms in the lower level processes that satisfy the implementation of the upper level axiom clause.

```
impl_local_axiom: OBLIGATION
    (FORALL (t1): IMPL(Axiom(t1)))
```

To prove the above obligations, the abstract machine axioms can be used in each lower level process. For example, to prove the impl_initial_state obligation, the initial clause of each lower level process can be asserted with the initial_state axiom.

Unlike the obligations that attempt to prove the mappings directly, the above obligations do not require a new axiom system, do not require an oracle for nondeterministic choices, and only use information about lower level processes.

VII.6.3. Correctness of Indirect Proof Obligations

The proof obligations for the parallel refinement mechanism as stated above are sufficient to show that for any invariant I that holds in the upper level, IMPL(I) holds in the lower level. Consider the correctness criteria (V), (C), (S), and (E) above. (V) is satisfied because by impl_initial_state, the values of the implementation of the variables in the lower level must be consistent with the values in the upper level. Variables in the upper level only change when a transition ends and at these times, the implementation of the variables in the lower level change consistently by impl_trans_exit. (C) is satisfied because the implementation of the variables in the lower level can only change value when the implementation of a transition ends by impl_vars_no_change. The forward direction of (S) is satisfied because whenever an upper level transition fires, a lower level transition will fire by impl_trans_fire. The reverse direction of (S) is satisfied because whenever the implementation of a transition fires in the lower level, its entry assertion holds by impl_trans_entry, it has been called by impl_trans_called, and no other transition is in the middle of execution by impl_trans_mutex. (E) is satisfied because (S) is satisfied and by impl_end1 and impl_end2, any time a start occurs, a corresponding end occurs and vice-versa.

More formally, any time an invariant I can be derived in the upper level, it is derived by a sequence of transformations from I to TRUE, $I \vdash_{f1/a1} I1 \vdash_{f2/a2} \ldots \vdash_{fn/an} TRUE$, where each transformation fi/ai corresponds to the application of a series fi of first-order logic axioms and a single abstract machine axiom ai. Since the implementation of each axiom of the ASTRAL abstract machine is preserved by the parallel refinement proof obligations, a corresponding proof at the lower level $IMPL(I) \vdash_{f1'/impl_a1} IMPL(I1) \vdash_{f2'/impl_a2} \ldots \vdash_{fn'/impl_an} TRUE$ can be constructed by replacing the application of each abstract machine axiom ai by impl_ai. Additionally, each series fi of first-order logic axioms is replaced by a series fi′ that takes any changes to the types of variables and constants into consideration.

VII.7. Proof Obligations for Mult_Add Circuit

This section shows the application of the parallel refinement proof obligations to the Mult_Add circuit.

```
impl_end1: OBLIGATION
    (FORALL (t1):
        past(A1.End(add, t1), t1) IMPLIES
            t1 ≥ 3)
```

By the entry assertion of add, multiply must end when add starts. The duration of add is 1 and the duration of multiply is 2, so the earliest add can end is at time 3. Thus, impl_end1 holds.

```
impl_end2: OBLIGATION
    (FORALL (t1):
        (past(M1.Start(multiply, t1 - 3), t1 - 3) AND
        past(M2.Start(multiply, t1 - 3), t1 - 3)) IFF
            past(A1.End(add, t1), t1))
```

For forward direction, it must be shown that add starts on A1 at t1 - 1. From the antecedent, multiply ends on both M1 and M2 at t1 - 1 so the entry assertion of add holds on A1 at time t1 - 1. A1 must be idle or else from the entry of add, multiply ended in the interval (t1 - 2, t1 - 1), which is not possible since multiply was still executing on M1 and M2 in that interval. Therefore, add starts at t1 - 1 on A1, thus ends at t1.

For the reverse direction, add starts on A1 at t1 - 1 from the antecedent. From the entry of add, multiply ends on both M1 and M2 at t1 - 1, so starts at t1 - 3. Thus, the reverse direction holds and impl_end2 holds.

```
impl_trans_entry: OBLIGATION
    (FORALL (t1):
        (past(M1.Start(multiply, t1), t1) AND
        past(M2.Start(multiply, t1), t1)) IMPLIES
            TRUE)
```

This formula trivially holds.

```
impl_trans_exit: OBLIGATION
    (FORALL (t1):
        past(A1.End(add, t1), t1) IMPLIES
            past(A1.sum, t1) = a * b + c * d)
```

By the exit assertion of add, past(A1.sum, t1) = past(M1.product, t1 - 1) + past(M2.product, t1 - 1). From the entry of add, multiply ends on both M1 and M2 at t1 - 1. By the exit assertion of multiply, past(M1.product, t1 - 1) = a * b and past(M2.product, t1 - 1) = c * d, so past(A1.sum, t1) = a * b + c * d. Thus, impl_trans_exit holds.

```
impl_trans_called: OBLIGATION
    (FORALL (t1):
        (past(M1.Start(multiply, t1), t1) AND
        past(M2.Start(multiply, t1), t1)) IMPLIES
            (EXISTS (t2):
                t2 ≤ t1 AND
                (past(M1.call(multiply, t2), t1) AND
                past(M2.call(multiply, t2), t1)) AND
                (FORALL (t3):
                    t2 ≤ t3 AND t3 < t1 IMPLIES
                        NOT (past(M1.Start(multiply, t3), t3) AND
                                past(M2.Start(multiply, t3), t3)))))
```

Since multiply started on M1 (M2) at time t1, by trans_called applied on process M1 (M2), multiply was called at some time t2 ≤ t1 and multiply has not started on M1 (M2) in the interval [t2, t1). By impl_call, the time that multiply was called on M1 and M2 must be the same. Thus, impl_trans_called holds.

```
impl_trans_mutex: OBLIGATION
    (FORALL (t1):
        (past(M1.Start(multiply, t1), t1) AND
        past(M2.Start(multiply, t1), t1)) IMPLIES
            (FORALL (t2):
                t1 < t2 AND t2 < t1 + 3 IMPLIES
                    NOT (past(M1.Start(multiply, t2), t2) AND
                            past(M2.Start(multiply, t2), t2))))
```

Since multiply started on M1 (M2) at time t1, by trans_mutex applied on process M1 (M2), nothing can fire on M1 (M2) until time t1 + 2. The multiply transition, however, is the only transition of M1 (M2) and multiply is not enabled until 1 time unit after the end of the last multiply, so cannot start until t1 + 3. Thus, impl_trans_mutex holds.

```
impl_trans_fire: OBLIGATION
    (FORALL (t1):
        (EXISTS (t2):
            t2 ≤ t1 AND
            (past(M1.call(multiply, t2), t1) AND
            past(M2.call(multiply, t2), t1)) AND
            (FORALL (t3):
                t2 ≤ t3 AND t3 < t1 IMPLIES
                    NOT (past(M1.Start(multiply, t3), t3) AND
                            past(M2.Start(multiply, t3), t3)))) AND
        (FORALL (t2):
            t1 - 3 < t2 AND t2 < t1 IMPLIES
                NOT (past(M1.Start(multiply, t2), t2) AND
                        past(M2.Start(multiply, t2), t2))) IMPLIES
        (past(M1.Start(multiply, t1), t1) AND
        past(M2.Start(multiply, t1), t1)))
```

To prove this obligation, it is first necessary to prove that M1.start(multiply) and M2.start(multiply) always occur at the same time. This can be proved inductively. At time 0, both M1 and M2 are idle. By impl_call, if multiply is called on either M1 or M2, multiply is called on both M1 and M2. If both are called, then both fire because the entry assertion of multiply is true since at time 0, multiply cannot have

ended. If neither is called, then neither can fire. For the inductive case, assume M1.start(multiply) and M2.start(multiply) have occurred at the same time up until time T0. Suppose multiply occurs on M1 (M2), then M1 (M2) was idle, multiply has been called since the last start, and it has been at least one time unit since multiply ended on M1 (M2). M2 (M1) cannot be executing multiply at T0 or else M1 (M2) must also be executing multiply by the inductive hypothesis, thus M2 (M1) must be idle. Similarly, it must have been at least one time unit since multiply ended on M2 (M1). By impl_call, multiply must have been called on M2 (M1) since it was called on M1 (M2). Thus, multiply is enabled on M2 (M1), so must fire. Therefore, M1.start(multiply) and M2.start(multiply) always occur at the same time. From this fact

```
(FORALL (t3):
    t2 ≤ t3 AND t3 < t1 IMPLIES
        NOT (past(M1.Start(multiply, t3), t3) AND
                past(M2.Start(multiply, t3), t3)))
```

is equivalent to

```
(FORALL (t3):
    t2 ≤ t3 AND t3 < t1 IMPLIES
        (NOT past(M1.Start(multiply, t3), t3) AND
        NOT past(M2.Start(multiply, t3), t3)))
```

Since nothing has started in the interval (t1 - 3, t1), nothing can end in the interval (t1 - 1, t1 + 2), thus the entry assertion of multiply on M1 is satisfied. Since the entry of multiply holds, multiply has been called but not yet serviced, and M1 is idle, multiply starts on M1 by trans_fire. Since multiply always starts on both M1 and M2 at the same time as shown above, impl_trans_fire holds.

```
impl_vars_no_change: OBLIGATION
    (FORALL (t1, t3):
        t1 ≤ t3 AND
        (FORALL (t2):
            t1 < t2 AND t2 ≤ t3 IMPLIES
                NOT past(A1.End(add, t2), t2) IMPLIES
                    (FORALL (t2):
                        t1 ≤ t2 AND t2 ≤ t3 IMPLIES
                            past(A1.sum, t1) = past(A1.sum, t2))))
```

This formula holds by the vars_no_change axiom applied on process A1.

```
impl_initial_state: OBLIGATION
    TRUE
```

This formula trivially holds.

```
impl_local_axiom: OBLIGATION
    TRUE
```

This formula trivially holds.

Since the proof obligations hold for the Mult_Add circuit, the lower level is a correct refinement of the upper level and thus the implementation of the upper level invariant, shown below, holds in the lower level.

```
FORALL t1: time, a, b, c, d: integer
    (   M1.Start(multiply(a, b), t1 - 3)
    &   M2.Start(multiply(c, d), t1 - 3)
    &   M1.Start(multiply, t1 - 3)
    &   M2.Start(multiply, t1 - 3)
→       FORALL t2: time
            (   t1 + dur1 ≤ t2
            &   t2 ≤ now
            →       past(A1.sum, t2) = a * b + c * d))
```

The previous example has shown that the parallel refinement mechanism can express the parallel implementation of a simple system in a simple and straightforward manner. More importantly, the proof obligations for a simple implementation were themselves simple. Now, the refinement of a much more complex system will be discussed along with the application of the proof obligations to it. From the following example, it will be shown that the parallel refinement mechanism can be used to express very complex parallel implementations, but at a cost of complicating the proofs of the proof obligations.

VII.8. Parallel Phone System

This section discusses the parallel refinement of a central control of a phone system. The phone system consists of a set of phones that need various services (e.g. getting a dial tone, processing digits entered into the phone, making a connection to the requested phone, etc.) as well as a set of central controls that perform the services. The system below is a modified version of the phone system defined in [CGK 97] that considers only local calls and not long distance calls.

The central control in [CGK 97] is not suitable for parallel refinement as it is already implemented in a sequential fashion. To make the central control more suitable, another level of abstraction will be added above the [CGK 97] level that allows a parallel refinement in addition to a sequential refinement. It will then be shown that the sequential refinement of [CGK 97] is just one possible implementation of the given system. The new top level of the central control process is given below. This definition and the corresponding refinements are only valid when a discrete time domain is used.

VII.8.1. Top Level of Central Control

The central control process has two transitions, Begin_Serve and Complete_Serve, each with duration serve_dur, where 2 * serve_dur is a divisor of the duration of every central control service. Begin_Serve and Complete_Serve execute cyclically and indicate the start and end, respectively, of execution of several different functions, each of which corresponds to a transition of the [CGK 97] central control (excluding the part referring to long distance calls). Since the goal is to allow different functions to be executed in

parallel, functions are allowed to begin in parallel in Begin_Serve and complete in parallel in Complete_Serve. Two different functions that begin service at the same time do not necessarily complete service at the same time, thus different functions can have different durations. The functions executed by Begin_Serve and Complete_Serve are:

(GDT) Give_Dial_Tones
(PD) Process_Digits
(PC) Process_Calls
(ER) Enable_Rings
(DRP) Disable_Ring_Pulses
(ERB) Enable_Ringbacks
(DRBP) Disable_Ringback_Pulses
(ST) Start_Talks
(TC) Terminate_Connections
(GA) Generate_Alarms

The functions are specified similarly to those in [CGK 97], but instead of specifying a separate transition for each, the entry assertion of Begin_Serve and the exit assertion of Complete_Serve are the conjunctions of the entry assertions of each function and the exit assertions of each function, respectively. In addition, each function is specified to service a set of phone processes instead of a single phone. Additionally, a variable "serving(phone): bool" is declared, which is true iff a phone has begun to be serviced but has not yet completed being serviced and is initially false for all phones. For each function g above, a set of phones W_g is defined, which is the set of phones waiting to be serviced by the function g. These sets are described as follows.

```
(W_GDT)   setdef P: phone (   P.Offhook & Phone_State(P) = Idle)
(W_PD)    setdef P: phone (   P.Offhook & Count(P) < 7
                          &  (   (   Phone_State(P) = Ready_To_Dial
                                 &   P.End(Enter_Digit) > End(Give_Dial_Tone(P)))
                             |   (   Phone_State(P) = Dialing
                                 &   P.End(Enter_Digit) > End(Process_Digit(P)))))
(W_PC)    setdef P: phone (   P.Offhook & Count(P) = 7
                          &   Phone_State(P) = Dialing
                          &   ~Get_ID(Number(P)).Offhook
                          &   Phone_State(Get_ID(Number(P)))= Idle)
...
```

In general, for each function g, the set W_g is the set of phones that satisfies the entry assertion of the transition of [CGK 97] associated with g. Let K_W_g be the maximum number of phones that can be served by the function g at any time and K_max be the maximum number of phones that can be served by any function at any time. Additionally, let Dur_g be the duration of the function g and Exit_g(P) be the exit assertion of the [CGK 97] transition associated with g applied to the phone P. In the following definitions of Begin_Serve and Complete_Serve, quantification over the functions g of the central control is used to simplify the presentation. The quantifiers can be expanded out over the 10 functions of the central control. For each function g, let serving_g be defined as setdef P: phone (serving(P) & P ISIN past(W_g, change(serving(P)) - serve_dur). That is, serving_g specifies the set of phones currently

being served by g. This definition is necessary since each W_g changes dynamically over time according to the behavior of the phone processes. Additionally, let serving_all be defined as setdef P: phone (serving(P)), which is the set of all phones being served.

Begin_Serve(S) is enabled for a nonempty set S when

- now is a multiple of 2 * serve_dur
- S is the union of the sets S_g, where for each function g,
  1. S_g only contains phones that are in W_g
  2. S_g does not contain any phones currently being served
  3. S_g ∪ serving_g contains at most K_W_g phones
- S ∪ serving_all contains at most K_max phones
- if S ∪ serving_all contains less than K_max phones, then for each function g, either S_g ∪ serving_g is at maximum capacity or all the phones of W_g are in S ∪ serving_all

The exit assertion of Begin_Serve specifies that the set of phones that begin being served is equal to the set parameter S.

```
Begin_Serve(S: nonempty_set_of_phone)
    ENTRY       [TIME:    serve_dur]
            now MOD (2 * serve_dur) = 0
        & EXISTS S_GDT, S_PD, ..., S_TC: set_of_phone
                (   S_GDT CONTAINED_IN W_GDT
                &   S_GDT SET_DIFF serving_all = S_GDT
                &   set_size(S_GDT UNION serving_GDT) ≤ K_W_GDT
                &   ...
                &   S = S_GDT UNION S_PD UNION ... UNION S_TC
                &   set_size(S UNION serving_all) ≤ K_max
                &   (   set_size(S UNION serving_all) < K_max
                →       FORALL g: central function
                            (   set_size(S_g UNION serving_g) = K_W_g
                            |   W_g CONTAINED_IN S_g UNION serving_all)))
    EXIT
            FORALL P: phone
                (   IF P ISIN S
                    THEN
                            serving(P)
                    ELSE
                            serving(P) ↔ serving′(P)
                    FI)
```

Complete_Serve is enabled when

- now + serve_dur is a multiple of 2 * serve_dur
- enough time has elapsed (i.e. the duration of the function) since some phone began being served such that service for that phone can be completed

The exit assertion of Complete_Serve specifies that all phones that have been served for at least the duration of the appropriate function will complete being served with the variables for each phone changed according to the exit assertion of the appropriate function.

```
Complete_Serve
    ENTRY       [TIME:    serve_dur]
            now MOD (2 * serve_dur) = serve_dur
        &   EXISTS P: phone, g: central function
                (   P ISIN serving_g
                &   now - change(serving(P)) + serve_dur ≥ Dur_g - serve_dur)
    EXIT
            FORALL P: phone, g: central function
                (   IF      P ISIN past(serving_g, now - serve_dur)
                        &   now - past(change(serving(P)), now - serve_dur) ≥
                                    Dur_g - serve_dur
                    THEN
                            Exit_g(P)
                        &   ~serving(P)
                    ELSE
                            serving(P) ↔ serving′(P)
                    FI)
```

Notice that the above specifications automatically define the updating of the set of phones. That is, each set W_g is updated according to changes in external processes (e.g. phones becoming offhook) and according to the changes made by the exit assertion of Complete_Serve.

VII.8.2. Sequential Refinement of Top Level Central Control

After redefining the top level specification of the central control, it becomes possible to show (assuming discrete time) that the original central control specification in [CGK 97] without long distance is one possible second level implementation of the top level given in section VII.8.1. Without loss of generality, it is assumed that the transitions of [CGK 97] only begin execution at times that are multiples of 2 * serve_dur. This essentially says that 2 * serve_dur is the fastest the system can recognize external changes. This can be accomplished by assuming the clause now MOD (2 * serve_dur) = 0 is conjoined to every entry assertion. The key to this refinement is mapping K_max to 1. This means that only a single function can occur at any given time in the system.

VII.8.2.1. IMPL Mapping

The IMPL mapping for the sequential refinement of the top level central control is shown below. K_max is mapped to 1 to force a sequential execution. The capacity of each function is mapped to 1 and the duration of each function is mapped to the duration of the corresponding transition of [CGK 97]. All other constants besides those below are mapped to themselves.

```
IMPL(K_max) == 1
IMPL(K_g) == 1 for all functions g
IMPL(Dur_g) == duration of transition corresponding to function g
```

A phone P being served in the upper level corresponds to the time between serve_dur after the start of some service transition and just before the end of that transition. All other variables besides serving are mapped to themselves.

```
IMPL(serving(P)) ==
        EXISTS tr: transition, t: time
            (   Start(tr(P), t)
            &   t + serve_dur ≤ now
            &   now < t + Duration(tr))
```

A start of Begin_Serve in the upper level occurs iff there is a start of a transition in the lower level at the same time. An end of Begin_Serve occurs iff there was a start of a transition serve_dur time units earlier.

```
IMPL(Start(Begin_Serve, now)) ==
        EXISTS tr: transition (Start(tr, now))
IMPL(End(Begin_Serve, now)) ==
        now ≥ serve_dur
    &   EXISTS tr: transition (Start(tr, now - serve_dur))
```

A start of Complete_Serve in the upper level occurs iff there was a start of a transition in the lower level at a time Duration(tr) - serve_dur time units earlier. An end of Complete_Serve occurs iff there is an end of a transition at the same time.

```
IMPL(Start(Complete_Serve, now)) ==
        EXISTS tr: transition
            (   now ≥ Duration(tr) - serve_dur
            &   Start(tr, now - Duration(tr) + serve_dur))
IMPL(End(Complete_Serve, now)) ==
        EXISTS tr: transition (End(tr, now))
```

VII.8.2.2. Proof of Sequential Refinement

The most interesting proof obligations in the sequential refinement of the top level central control are the impl_trans_entry and impl_trans_exit obligations. In these proof obligations, let W_g, serving_g, and serving_all refer to IMPL(W_g), IMPL(serving_g), and IMPL(serving_all), respectively.

VII.8.2.2.1. Impl_trans_entry Obligation

In the Begin_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
    (past
        (   EXISTS tr: transition (Start(tr, now))
    →       EXISTS S: nonempty_set_of_phone
                (   now MOD (2 * serve_dur) = 0
                &   EXISTS S_GDT, S_PD, ..., S_TC: set_of_phone
                        (   S_GDT CONTAINED_IN W_GDT
                        &   S_GDT SET_DIFF serving_all = S_GDT
                        &   set_size(S_GDT UNION serving_GDT) ≤ 1
                        &   ...
                        &   S = S_GDT UNION S_PD UNION ... UNION S_TC
                        &   set_size(S UNION serving_all) ≤ 1
                        &   (   set_size(S UNION serving_all) < 1
                        →       FORALL g: central function
                                    (   set_size(S_g UNION serving_g) = 1
                                    |   W_g CONTAINED_IN S_g UNION serving_all)))), t1))
```

By the antecedent, there is some transition tr_g that starts at time t1. Let P be the phone that tr_g is servicing. The existential clause of the consequent is satisfied by the set consisting of only P. By previous assumption, the transitions of [CGK 97] can only start at times that are multiples of 2 * serve_dur, thus the first conjunct of the consequent holds.

The implementation of serving only holds when a transition is in the middle of execution and serve_dur has elapsed since the transition fired. By trans_mutex, there can only be one such transition. The only transition in the middle of execution is tr_g and at t1, serve_dur time has not yet elapsed. Therefore, set_size(serving_all) = 0. The second conjunct of the consequent is satisfied by the collection of sets S_h, where S_h contains only P for h = g and is empty otherwise by the entry assertion of tr_g.

In the Complete_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
   (past
      (   EXISTS tr: transition
             (   now ≥ Duration(tr) - serve_dur
             &   Start(tr, now - Duration(tr) + serve_dur))
   →     (   now MOD (2 * serve_dur) = serve_dur
         &   EXISTS P: phone, tr1: transition
                (   P ISIN serving_g
                &   now - change(
                              EXISTS tr: transition, t: time
                                 (   Start(tr(P), t)
                                 &   t + serve_dur ≤ now
                                 &   now < t + Duration(tr))) + serve_dur ≥
                                          Duration(tr1) - serve_dur)), t1))
```

By previous assumption, transitions only start at times that are multiples of 2 * serve_dur and have durations that are multiples of 2 * serve_dur, thus t1 - Duration(tr) is a multiple of 2 * serve_dur and t1 - Duration(tr) + serve_dur MOD (2 * serve_dur) = serve_dur. Therefore, the first conjunct holds.

Let tr_g be the transition that fires at t1 - Duration(tr_g) + serve_dur. Let P be the phone that tr_g is servicing. At t1, tr_g has not yet ended and a serve_dur has elapsed since tr_g began, thus the first part of the existential clause holds.

The implementation of serving changes whenever serve_dur has elapsed since the start of a transition or at the end of a transition. Since tr_g is still executing, the last change is at the start time of tr_g + serve_dur or t1 - Duration(tr_g) + 2 * serve_dur. Thus, t1 - (t1 - Duration(tr_g) + 2 * serve_dur) + serve_dur ≥ Duration(tr_g) - serve_dur since Duration(tr_g) - serve_dur ≥ Duration(tr_g) - serve_dur. Thus, the second part of the existential clause holds.

VII.8.2.2.2. Impl_trans_exit Obligation

In the Begin_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
    (past
        (   now ≥ serve_dur
        &   EXISTS tr: transition (Start(tr, now - serve_dur))
    →       EXISTS S: nonempty_set_of_phone
                (FORALL P: phone
                    (   IF P ISIN S
                        THEN
                                EXISTS tr: transition, t: time
                                    (   Start(tr(P), t)
                                    &   t + serve_dur ≤ now
                                    &   now < t + Duration(tr))
                        ELSE
                                EXISTS tr: transition, t: time
                                    (   Start(tr(P), t)
                                    &   t + serve_dur ≤ now
                                    &   now < t + Duration(tr))
                            ↔ EXISTS tr: transition, t: time
                                    (   past(Start(tr(P), t), now - serve_dur)
                                    &   t + serve_dur ≤ now - serve_dur
                                    &   now - serve_dur < t + Duration(tr)))), t1))
```

By the antecedent, there is some transition tr_g that starts at time t1 - serve_dur. Let P be the phone that tr_g is servicing. The existential clause of the consequent is satisfied by the set consisting of only P. Only one phone can satisfy the setdef predicate in the consequent. P satisfies the predicate for transition tr_g and time t1 - serve_dur because Start(tr_g(P), t1 - serve_dur) from the antecedent, t1 - serve_dur + serve_dur ≤ t1, and t1 < t1 - serve_dur + Duration(tr_g) since Duration(tr_g) must be a multiple of 2 * serve_dur.

In the Complete_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
    (past
        (   EXISTS tr: transition (End(tr, now))
    →       FORALL P: phone, g: central function
                (   IF      P ISIN past(serving_g, now - serve_dur)
                    &   now - past(change(
                                    EXISTS tr: transition, t: time
                                        (   Start(tr(P), t)
                                        &   t + serve_dur ≤ now
                                        &   now < t + Duration(tr))), now - serve_dur) ≥
                                            Dur_g - serve_dur
                THEN
                        Exit_g(P)
                    &   ~EXISTS tr: transition, t: time
                            (   Start(tr(P), t)
                            &   t + serve_dur ≤ now
                            &   now < t + Duration(tr))
```

```
ELSE
        EXISTS tr: transition, t: time
            (   Start(tr(P), t)
            &   t + serve_dur ≤ now
            &   now < t + Duration(tr))
    ↔ EXISTS tr: transition, t: time
            (   past(Start(tr(P), t), now - serve_dur)
            &   t + serve_dur ≤ now - serve_dur
            &   now - serve_dur < t + Duration(tr))
FI), t1))
```

Let tr_g be the transition that ends at t1 for a phone P. There can only be one phone and transition for which the if condition is satisfied, since only only one transition can be in the middle of execution at any given time. The if condition is satisfied for phone P and function g of tr_g.

At t1 - serve_dur, the last change of the implementation of serving is at t1 - Duration(tr_g) + serve_dur, so the first part of the condition holds since past(W_g, t1 - Duration(tr_g) + serve_dur - serve_dur) holds by trans_entry. The second part of the condition holds since t1 - (t1 - Duration(tr_g) + serve_dur) ≥ Duration(tr_g) - serve_dur.

P is the only phone for which the then branch must hold. The exit of tr_g holds for P by trans_exit. Since tr_g ends at t1, the implementation of serving no longer holds at t1, thus the then branch holds. For all other phones, the else branch must hold. Both existential clauses are false because no phone other than P was being serviced at t1 - serve_dur and no phone can be serve_dur into its execution at t1 since tr_g just ended at t1. Thus, the then branch holds.

VII.8.3. Parallel Refinement of Top Level Central Control

In the parallel refinement of the top level central control, the central control is refined into several parallel processes, each of which is devoted to a single function g of the top level central control. Each one of these processes executes two transitions that correspond to Begin_Serve and Complete_Serve at the top level.

The main issue in this step is the mapping of the global state of the central control into disjoint components to be assigned to the different lower level parallel processes. Figure VII.8.3-1 shows the relationship between the functions and the variables of the central control. A function connected to a variable indicates that the exit assertion of the function sets the variable.

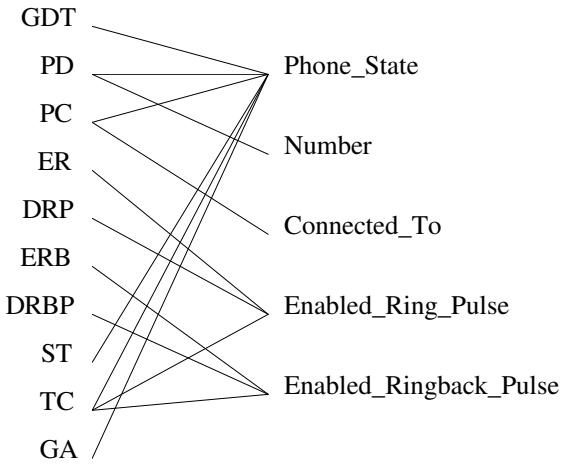

Figure VII.8.3-1: Function and variable relationship in the central control

Note that the exit assertions of the GDT and PD functions have been modified to allow Number to be exclusive to the PD function. The Number(P) reference was removed from the GDT exit assertion and the exit assertion of PD was changed to the following.

```
IF Phone_State′(P) = Ready_To_Dial
THEN
        Number(P) BECOMES LISTDEF(P.Next_Digit′)
    &   Phone_State(P) BECOMES Dialing
ELSE
        Number(P) BECOMES Number′(P) CONCAT LISTDEF(P.Next_Digit′)
FI
```

The most critical variable of the central control is the Phone_State variable, which is set by 6 of the 10 functions. In the following discussion of the parallel refinement of the top level central control, only the implementation of the Phone_State variable and its corresponding type Enabled_State will be described. The other variables can be mapped in a similar fashion.

The type Enabled_State describes all states through which the managing of a call passes during the evolution of the call itself. Such states build a sort of “chain” and the various state transformations move the state of each phone call through it from Idle to Ready_To_Dial, etc. In the implementation of the top level, each step of this path is executed by a different process. This causes a difficulty since all processes must have disjoint states and shared variables (in writing) are not allowed. A possible systematic implementation schema (perhaps not optimal) to split the original state space into disjoint components is to map the type Enabled_State into a structure of time fields, where each enumerated constant in Enabled_State has a field in the structure as shown below.

```
IMPL(Enabled_State) ==
        STRUCTURE OF (Idle, Ready_To_Dial, Ringing, ...: time)
```

The basic idea of this mapping is that each field of the structure is a timestamp and the field with the most recent timestamp determines the value of variables of the structure type. Since Enabled_State is not compatible with IMPL(Enabled_State), a mapping for constants must be defined as shown below.

```
IMPL(v_es: Enabled_State) ==
    CASE v_es OF
        Idle:
            choose i_v_es: IMPL(Enabled_State)
                (   FORALL f: field (i_v_es[f] = 0)
                |   FORALL f: field
                        (f ≠ Idle → i_v_es[Idle] > i_v_es[f]))
        Ready_To_Dial:
            choose i_v_es: IMPL(Enabled_State)
                (   FORALL f: field
                        (f ≠ Ready_To_Dial → i_v_es[Ready_To_Dial] > i_v_es[f]))
        Ringing:
            choose i_v_es: IMPL(Enabled_State)
                (   FORALL f: field
                        (f ≠ Ringing → i_v_es[Ringing] > i_v_es[f]))
        ...
    ESAC
```

This mapping states that a constant that is Idle in the upper level maps to a structure of type IMPL(Enabled_State) such that all the fields are 0 or the Idle field is greater than all the other fields. For values v other than Idle, a constant maps to a structure such that the field associated with v is greater than all the other fields.

In addition to a mapping for constants of type Enabled_State, a mapping must also be defined for the operators with operands of type Enabled_State. The only operator used on operands of type Enabled_State is the = operator. The mapping for the = operator is shown below.

```
IMPL(=(v_es1, v_es2: Enabled_State): bool) ==
    (   (   FORALL f1: field (IMPL(v_es1)[f1] = 0)
    →       FORALL f1: field
                (   IMPL(v_es2)[f1] = 0
                |   (f1 ≠ Idle → IMPL(v_es2)[Idle] > IMPL(v_es2)[f1])))
    |   (   FORALL f1: field (IMPL(v_es2)[f1] = 0)
    →       FORALL f1: field
                (   IMPL(v_es1)[f1] = 0
                |   (f1 ≠ Idle → IMPL(v_es1)[Idle] > IMPL(v_es1)[f1])))
    |   (   EXISTS f1: field
                (   FORALL f2: field
                        (   f1 ≠ f2
                    →       (   IMPL(v_es1)[f1] > IMPL(v_es1)[f2]
                            &   IMPL(v_es2)[f1] > IMPL(v_es2)[f2])))))
```

This mapping states that two constants of type Enabled_State in the upper level are equal iff either (1) all the fields in the structure generated from the implementation of one of the constants are 0 and the other structure is either all 0's or the Idle field is greater than all the other fields or (2) there is a field that is greater than all the other fields in both structures in the lower level.

For the implementation of Phone_State, each server has a variable "f(phone): time", for each field f of the IMPL(Enabled_State) structure that the server is responsible for. The mapping for Phone_State is shown below.

```
IMPL(Phone_State(P)) ==
        choose v_es: IMPL(Enabled_State)
            (   v_es[Idle] = TC.Idle(P)
            &   v_es[Ready_To_Dial] = GDT.Ready_To_Dial(P)
            &   ...)
```

This mapping specifies that the state of a phone P is determined by the server that has most recently timestamped a field of P. Thus, it is possible for all the servers to directly affect the state of a phone. Figure VII.8.3-2 shows the mapping from the servers of the lower level to the values of Enabled_State. Note that the value "Calling" is not mapped to any server because Calling is only used for long distance calls. For this mapping to work, it must be guaranteed that no two servers ever give the same timestamp to the same phone. This is a problem, for example, if a phone is offhook and a "slow" server begins to serve the phone and then while this is occurring, the user of the phone hangs up, and the TC server attempts to set the phone to Idle.

This problem can be avoided by keeping the Begin_Serve/Complete_Serve mechanism of the top level. Each server will only attempt to serve a phone if no other server is serving that phone. No two servers will ever be able to execute Begin_Serve at the same time for the same phone because at any given time, there is a unique function that a phone needs next. Since discrete time is assumed, it can also be guaranteed that Begin_Serve's cannot overlap on different servers. Thus, no two servers can ever give the same timestamp to the same phone.

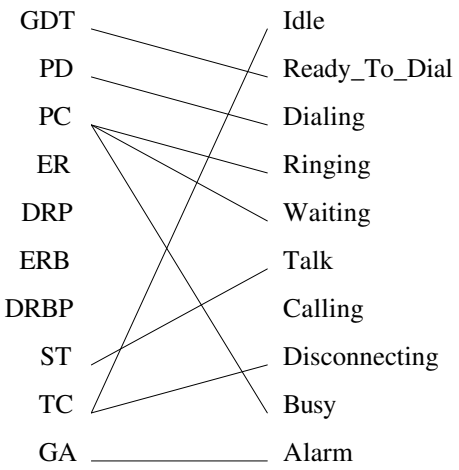

Figure VII.8.3-2: Mapping from servers to Enabled_State values

For the implementation of serving, each server has a variable "serving_set: set_of_phone". Instead of storing the value of serving for every phone, a server only needs to store those phones that it is currently serving. The mapping for serving is shown below.

```
IMPL(serving(P)) == EXISTS SP: server (P ISIN SP.serving_set)
```

The value of K_max in the upper level is intended to limit the amount of parallelism in the system for sequential implementations. In the parallel refinement, it is undesirable to place any such limitation on the number of phones that can be served systemwide. Thus, the mapping for K_max is the maximum allowable parallelism sum_K, where sum_K = $\sum_g K_W_g$.

```
IMPL(K_max) == sum_K
```

All other constants map to themselves.

In the top level, Begin_Serve is enabled when there exists a set of phones S that can be partitioned into 10 disjoint subsets such that there is a subset S_g for each function g that is limited in size by K_W_g - set_size(serving_g) and must contain only the phones that begin being served. In the second level, each server has a Begin_Serve transition that is enabled when there exists a set of phones that corresponds to the disjoint subset S_g for the function g that the server performs. The exit assertion of Begin_Serve on a server SP_g specifies that the set of phones that begin being served by SP_g is equal to the disjoint subset S_g. The definition of Begin_Serve for the PD server is shown below.

```
Begin_Serve(S: nonempty_set_of_phone)
    ENTRY       [TIME:    serve_dur]
            now MOD (2 * serve_dur) = 0
        &   S CONTAINED_IN W_PD
        &   S SET_DIFF serving_all = S
        &   set_size(S UNION serving_set) ≤ K_W_PD
        &   (   set_size(S UNION serving_set) < K_W_PD
        →       W_PD CONTAINED_IN S UNION serving_all)
    EXIT
            serving_set = serving_set' UNION S
```

A start of Begin_Serve in the upper level corresponds to a start of Begin_Serve on some server in the lower level. The mapping for an end of Begin_Serve is similarly defined.

```
IMPL(Start(Begin_Serve, now)) ==
        EXISTS SP: server
            (SP.Start(Begin_Serve, now))
IMPL(End(Begin_Serve, now)) ==
        EXISTS SP: server
            (SP.End(Begin_Serve, now))
```

The definition of Complete_Serve for servers in the lower level is similar to the definition of Complete_Serve in the upper level except that each server SP_g only checks the phones it is serving when determining which phones have been being served for Dur_g and changes the state of these phones according to Exit_g. The definition of Complete_Serve for the PD server is shown below.

```
Complete_Serve
    ENTRY       [TIME:    serve_dur]
            now MOD (2 * serve_dur) = serve_dur
        &   EXISTS P: phone
                (   P ISIN serving_set
                &   now - change(P ISIN serving_set) + serve_dur ≥ Dur_PD - serve_dur)
```

```
EXIT
    FORALL P: phone
      (  IF  (   P ISIN serving_set′
             &   now - past(change(P ISIN serving_set), now - serve_dur) ≥
                     Dur_PD - serve_dur)
          THEN
                 P ~ISIN serving_set
             &   IF FORALL SP: server
                        (   SP ≠ GDT
                     →      GDT.Ready_To_Dial(P) > SP.ES(P))
                 THEN
                        Number(P) BECOMES LISTDEF(P.Next_Digit′)
                     &  Dialing(P) = now
                 ELSE
                        Number(P) BECOMES
                              Number′(P) CONCAT LISTDEF(P.Next_Digit′)
                 FI
          ELSE
                 P ISIN serving_set′ ↔ P ISIN serving_set
          FI)
```

When the state of a phone P was previously Ready_To_Dial, the new state of P is set to Dialing, since the timestamp of Dialing(P) is set to the current time by the expression "Dialing(P) = now". Note that the expression SP.ES(P) is a convenience notation to refer to the timestamp of phone P for any variables that are components of the Phone_State mapping in the server SP (e.g. TC.Idle(P)).

A start of Complete_Serve in the upper level corresponds to a start of Complete_Serve on some server in the lower level. The mapping for an end of Complete_Serve is similarly defined.

```
IMPL(Start(Complete_Serve, t)) ==
    EXISTS SP: server
        (SP.Start(Complete_Serve, t))
IMPL(End(Complete_Serve, t)) ==
    EXISTS SP: server
        (SP.End(Complete_Serve, t))
```

VII.8.4. Proof of Parallel Refinement of Top Level Central Control

The proof obligations that will be shown for the parallel refinement of the top level of the central control are the impl_trans_entry obligation for Begin_Serve, the impl_trans_exit obligation for Complete_Serve, and the impl_trans_fire obligation. In these proof obligations, let W_g, serving_g, and serving_all refer to IMPL(W_g), IMPL(serving_g), and IMPL(serving_all), respectively.

VII.8.4.1. Impl_trans_entry Obligation for Begin_Serve

```
FORALL t1: time
   (past
      (   EXISTS SP: server (SP.Start(Begin_Serve, now))
   →      EXISTS S: nonempty_set_of_phone
             (   now MOD (2 * serve_dur) = 0
             &   EXISTS S_GDT, S_PD, ..., S_TC: set_of_phone
                    (   S_GDT CONTAINED_IN W_GDT
                    &   S_GDT SET_DIFF serving_all = S_GDT
                    &   set_size(S_GDT UNION GDT.serving_set) ≤ K_W_GDT
                    &   ...
                    &   S = S_GDT UNION S_PD UNION ... UNION S_TC
                    &   set_size(S UNION serving_all) ≤ sum_K
                    &   (   set_size(S UNION serving_all) < sum_K
                    →       FORALL g: central function
                               (   set_size(S_g UNION SP_g.serving_set) = K_W_g
                               |   W_g CONTAINED_IN S_g UNION serving_all)))), t1))
```

Begin_Serve can only be enabled at times that are multiples of 2 * serve_dur. Complete_Serve can only be enabled at times that are serve_dur plus a multiple of 2 * serve_dur. Begin_Serve and Complete_Serve each have duration serve_dur, thus any time Begin_Serve fires on some server, every other server will either be idle or will fire Begin_Serve. At t1, Begin_Serve fires on some server, thus every other server is either idle or fires Begin_Serve. For each server SP_g on which Begin_Serve fires at t1, let S2_g be the set parameter for which Begin_Serve fired. The consequent is satisfied by the union of all sets S2_g.

The first part of the consequent is satisfied by the entry of Begin_Serve on a server on which it started. The second part is satisfied by S_g = S2_g for the servers SP_g that Begin_Serve started on at t1 and by S_g = ∅ for the other servers. By the entry of Begin_Serve, S2_g CONTAINED_IN W_g, S2_g SET_DIFF serving_all = S2_g, and set_size(S_g UNION SP_g.serving_set) ≤ K_W_g. The empty set trivially satisfies these constraints. By the definition of S, S = UNION S2_g = UNION S2_g UNION EMPTY. Each S_g UNION serving_g must be ≤ K_W_g by the entry of Begin_Serve, so set_size(S UNION serving_all) must be ≤ sum_K. Suppose set_size(S_g UNION serving_g) < K_W_g & W_g ~CONTAINED_IN S UNION serving_all for some g. Let P be a phone that needs service from g, but is not in S_g UNION serving_g. Suppose SP_g starts Begin_Serve at t1. In this case, S2_g does not satisify the entry assertion of Begin_Serve because set_size(S2_g) < K_W_g and yet P is in W_g, but not being served by any other server. Therefore, ~SP_g.Start(Begin_Serve(S2_g), t1), which is a contradiction. Suppose SP_g did not start Begin_Serve at t1. In this case, the entry assertion holds because P needs service and SP_g still has capacity left. SP_g must be idle because Begin_Serve and Complete_Serve are mutually exclusive by their entry assertions. Thus, SP_g.Start(Begin_Serve, t1), which is also a contradiction.

VII.8.4.2. Impl_trans_exit Obligation for Complete_Serve

```
FORALL t1: time
    (past
        (   EXISTS SP: server
                (SP.End(Complete_Serve, now))
    →       FORALL P: phone, g: central function
                (   IF      P ISIN past(SP_g.serving_set, now - serve_dur)
                    &   now - past(change(EXISTS SP: server (P ISIN SP.serving_set)),
                                now - serve_dur) ≥ Dur_g - serve_dur
                    THEN
                            Exit_g(P)
                    &   ~EXISTS SP: server (P ISIN SP.serving_set)
                    ELSE
                            EXISTS SP: server (P ISIN SP.serving_set)
                        ↔ past(EXISTS SP: server (P ISIN SP.serving_set), now - serve_dur)
                    FI), t1))
```

Suppose there is some phone P and function g such that the if condition holds, but the then branch does not hold. Complete_Serve must be enabled on SP_g at t1 - serve_dur because now MOD (2 * serve_dur) = serve_dur by the entry assertion of Complete_Serve on a server on which it ended at t1, and P ISIN SP_g.serving_set at t1 - serve_dur and t1 - serve_dur - change(P ISIN serving_set) + serve_dur ≥ Dur_PD - serve_dur by the if condition. SP_g must be idle because Begin_Serve and Complete_Serve are mutually exclusive by their entry assertions. By trans_fire, Complete_Serve starts at t1 - serve_dur on SP_g, thus its exit assertion holds at t1, so the then branch holds.

For the else branch, suppose a phone P and central function g do not satisfy the if condition, but the status of P in SP_g.serving_set changes at t1. Begin_Serve and Complete_Serve are mutually exclusive, thus Complete_Serve on SP_g changes the status. Complete_Serve on SP_g can only change the status, however, when the if condition holds for P, which is a contradiction.

VII.8.4.3. Impl_trans_fire Obligation

```
FORALL t1: time
    (   past
            (EXISTS S: nonempty_set_of_phone
                (   now MOD (2 * serve_dur) = 0
                &   EXISTS S_GDT, S_PD, ..., S_TC: set_of_phone
                        (   S_GDT CONTAINED_IN W_GDT
                        &   S_GDT SET_DIFF serving_all = S_GDT
                        &   set_size(S_GDT UNION GDT.serving_set) ≤ K_W_GDT
                        &   ...
                        &   S = S_GDT UNION S_PD UNION ... UNION S_TC
                        &   set_size(S UNION serving_all) ≤ sum_K
                        &   (   set_size(S UNION serving_all) < sum_K
                        →       FORALL g: central function
                                    (   set_size(S_g UNION SP_g.serving_set) = K_W_g
                                    |   W_g CONTAINED_IN S_g UNION serving_all)))), t1)
```

```
    &  FORALL t2: time
          (   t1 - serve_dur < t2 & t2 < t1
       →  (   ~EXISTS SP: server
                 (past(SP.Start(Begin_Serve, t2), t2))
          &   ~EXISTS SP: server
                 (past(SP.Start(Complete_Serve, t2), t2))))
→      EXISTS SP: server
          (past(SP.Start(Begin_Serve, t1))))
```

Let S be a set of phones satisfying the existential clause in the antecedent. Let S_g be a nonempty set of the second part of the existential clause. There must be such a set since S in nonempty and S is the union of all such sets. The entry assertion of Begin_Serve is satisfied by the set S_g on SP_g at t1. By the antecedent, no server is executing any transition at t1, thus Begin_Serve will fire on SP_g at t1 by trans_fire.

```
FORALL t1: time
   (   past
          (   now MOD (2 * serve_dur) = serve_dur
          &   EXISTS P: phone, g: central function
                 (   P ISIN SP_g.serving_set
                 &   now - change(EXISTS SP: server (P ISIN SP.serving_set)) +
                           serve_dur ≥ Dur_g - serve_dur), t1)
   &  FORALL t2: time
          (   t1 - serve_dur < t2 & t2 < t1
       →  (   ~EXISTS SP: server
                 (past(SP.Start(Begin_Serve, t2), t2))
          &   ~EXISTS SP: server
                 (past(SP.Start(Complete_Serve, t2), t2))))
→      EXISTS SP: server
          (past(SP.Start(Complete_Serve, t1))))
```

Let P and g be the phones satisfying the existential clause in the antecedent. Thus, P is in the serving set of SP_g at t1. Also, P has been being served for Dur_g - 2 * serve_dur. Thus, the entry assertion of Complete_Serve on SP_g holds at t1. By the antecedent, no server is executing any transition at t1, thus Complete_Serve will fire on SP_g at t1 by trans_fire.

VII.8.5. Parallel Refinement of Second Level Process Call Server

This section discusses the parallel refinement of the second level process call server. The other servers of the second level central control can be refined in a similar manner. The PC server is implemented by a parallel array of K_W_PC *microservers*, where each microserver is devoted to processing the calls of a single phone. Each microserver picks a phone from W_PC according to some possibly nondeterministic policy and inserts its identifier into a set of served phones through a sequence of two transitions. The union of the elements of such sets over all the PC microservers implements the serving set of the upper level PC server.

At this refinement level, it is not possible to statically allocate the individual phone timestamps of Ringing, Waiting, and Busy to different microservers or else there would be no way for phones allocated

to the same microserver to be serviced at the same time, which is possible at the higher levels. Instead, microservers dynamically hold the state of the set of phones that were last serviced on that microserver. To control the size of the served set, a microserver removes a set of phones from the set such that each phone is in the set if the timestamp for that phone on some other macroserver has changed more recently than the phone was added to the served set.

The PC microservers process phones in pairs, where each pair is of the type "phone_pair: STRUCTURE OF (Waiting: phone, Ringing: phone)". The variable "serving: boolean" specifies if the microserver is currently serving a pair of phones. The variable "serving_pair: phone_pair" specifies the pair of phones the micoserver is going to connect. Finally, the variable "served_set: list of phone_pair" specifies the set of phone pairs whose calls have been processed by the microserver, but whose state has not yet been changed by any of the other macroservers.

The timestamp that a phone P became waiting is the time that P became the waiting phone of a phone pair in the served_set of some microserver. The waiting timestamp of P is 0 if no such phone pair exists.

```
IMPL(Waiting(P)) ==
    IF    EXISTS MSP: microserver, PP: phone_pair
            (   PP ISIN MSP.served_set
            &   PP[Waiting] = P
            &   PP[Ringing] ≠ P)
    THEN
        change(EXISTS MSP: microserver, PP: phone_pair
                (   PP ISIN MSP.served_set
                &   PP[Waiting] = P
                &   PP[Ringing] ≠ P))
    ELSE
        0
    FI
```

The timestamp that a phone P became ringing is the time that P became the ringing phone of a phone pair in the served_set of some microserver. The ringing timestamp of P is 0 if no such phone pair exists.

```
IMPL(Ringing(P)) ==
    IF    EXISTS MSP: microserver, PP: phone_pair
            (   PP ISIN MSP.served_set
            &   PP[Ringing] = P
            &   PP[Waiting] ≠ P)
    THEN
        change(EXISTS MSP: microserver, PP: phone_pair
                (   PP ISIN MSP.served_set
                &   PP[Ringing] = P
                &   PP[Waiting] ≠ P))
    ELSE
        0
    FI
```

The timestamp that a phone P became busy is the time that P became both the waiting phone and the ringing phone of a phone pair in the served_set of some microserver. The busy timestamp of P is 0 if no such phone pair exists.

```
IMPL(Busy(P)) ==
    IF    EXISTS MSP: microserver, PP: phone_pair
            (   PP ISIN MSP.served_set
            &   PP[Waiting] = P
            &   PP[Ringing] = P)
    THEN
        change(EXISTS MSP: microserver, PP: phone_pair
                (   PP ISIN MSP.served_set
                &   PP[Waiting] = P
                &   PP[Ringing] = P))
    ELSE
        0
    FI
```

A phone is connected to a phone P if it is in a phone pair with P on some microserver. Connected_To is set to P otherwise.

```
IMPL(Connected_To(P)) ==
    IF
        EXISTS P2: phone
            (   P2 ≠ P
            &   EXISTS MSP: microserver, PP: phone_pair
                    (   PP ISIN MSP.served_set
                    &   (   PP[Waiting] = P & PP[Ringing] = P2
                        |   PP[Waiting] = P2 & PP[Ringing] = P)))
    THEN
        choose P2: phone
            (   P2 ≠ P
            &   EXISTS MSP: microserver, PP: phone_pair
                    (   PP ISIN MSP.served_set
                    &   (   PP[Waiting] = P & PP[Ringing] = P2
                        |   PP[Waiting] = P2 & PP[Ringing] = P)))
    ELSE
        P
    FI
```

The implementation of the serving set is the set of phones that are waiting the waiting phone in the serving pair of some microserver that is serving.

```
IMPL(serving_set) ==
    setdef P: phone
       (EXISTS MSP: microserver
            (   MSP.serving
            &   MSP.serving_pair[Waiting] = P))
```

Each PC server has two transitions, which correspond to Begin_Serve and Complete_Server in the upper level. Begin_Serve finds a pair of phones in W_PC to be connected. Complete_Serve commits the connection between the two phones identified in the preparation phase and resets the state of all phones in

its list Served_Phones that have been serviced more recently by some other upper level macroserver. The definitions of Begin_Serve and Complete_Serve are given below.

```
Begin_Serve(P: phone)
    ENTRY       [TIME:    serve_dur]
            now MOD (2 * serve_dur) = 0
        &   ~serving
        &   P ISIN W_PD
        &   P ~ISIN serving_all
        &   set_size(setdef P2: phone (P2 ISIN W_PD & P2 ~ISIN serving_all & P2 < P))
                =   set_size(setdef MSP: microserver (~MSP.serving & MSP < self))
    EXIT
            serving
        &   serving_pair = choose PP: phone_pair
                                (   PP[Waiting] = P
                                &   PP[Ringing] =
                                        IF      GET_ID(PD.Number(P)).Offhook
                                            |   EXISTS SP: server
                                                    (   SP ≠ TC
                                                →       SP.ES(GET_ID(PD.Number(P))) >
                                                                TC.Idle(GET_ID(PD.Number(P))))
                                            |   EXISTS g: central function
                                                    (GET_ID(PD.Number(P)) ISIN W_g)
                                        THEN
                                                P
                                        ELSE
                                                GET_ID(PD.Number(P))
                                        FI)
```

The last conjunct of the entry assertion states that if there are n phones satisfying the condition whose IDs are less than P's ID, then there exist n microservers whose IDs are less than self and are available. This is a simple trick to state that the available microservers are allocated in order of increasing ID number to phones that need their service; in such a way we avoid conflict and can easily prove requirements on the number of phone calls that will be served. A start of Begin_Serve in the upper level corresponds to a start of Begin_Serve on some microserver in the lower level. The mapping for an end of Begin_Serve is similarly defined.

```
IMPL(Start(Begin_Serve, now)) ==
        EXISTS MSP: microserver
            (MSP.Start(Begin_Serve, now))
IMPL(End(Begin_Serve, now)) ==
        EXISTS MSP: microserver
            (MSP.End(Begin_Serve, now))
```

Complete_Serve finishes serving the phones in serving_pair.

```
Complete_Serve
    ENTRY       [TIME:    Dur_PC - serve_dur]
            serving
    EXIT
            FORALL PP: phone_pair
                (   IF  (   PP ISIN served_set′
                        &   EXISTS SP: server
                                (SP.ES(PP[Waiting]) >
                                    past(change(PP ISIN served_set),
                                            now - Dur_PC + serve_dur)
                        &   EXISTS SP: server
                                (SP.ES(PP[Ringing]) >
                                    past(change(PP ISIN served_set),
                                            now - Dur_PC + serve_dur)
                    THEN
                            PP ~ISIN served_set
                    ELSE
                            PP ISIN served_set′ ↔ PP ISIN served_set
                    FI)
```

Notice that the duration of Complete_Serve is now Dur_PC - serve_dur, which is the time it takes to complete processing a call, whereas in the higher levels, the duration was a small duration, serve_dur, so that phones could complete being serviced at almost any time. Also note that it should be possible to bound the number of phones that any microserver actually has to delete from its serving list in any execution of Complete_Serve as well as the size of the served set based on Dur_g and K_W_g of all macroservers g.

VII.8.6. Proof of Parallel Refinement of Process Call Server

The proof obligations that will be shown for the parallel refinement of the process call server are the impl_trans_mutex and impl_vars_no_change obligations.

VII.8.6.1. Impl_trans_mutex Obligation

In the Begin_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
    (   past(EXISTS MSP: microserver
                (MSP.Start(Begin_Serve, t1)), t1)
→   (   ~past(EXISTS MSP: microserver
                (MSP.Start(Complete_Serve, t1)), t1)
    &   FORALL t2: time
                (t1 < t2 & t2 < t1 + serve_dur
            →       ~past(EXISTS MSP: microserver
                            (MSP.Start(Begin_Serve, t2)), t2))
    &   FORALL t2: time
                (t1 < t2 & t2 < t1 + serve_dur
            →       ~past(EXISTS MSP: microserver
                            (MSP.Start(Complete_Serve, t2)), t2))))
```

Begin_Serve can only start at times that are multiples of 2 * serve_dur by its entry assertion. Complete_Serve is enabled when serving holds. Begin_Serve sets serving and Complete_Serve resets

serving so Complete_Serve can only start immediately when a Begin_Serve ends. Therefore, Complete_Serve can only start at times that are serve_dur after a multiple of 2 * serve_dur. Since a Begin_Serve starts at t1, Complete_Serve cannot have started on any microserver in the interval (t1 - serve_dur, t1 + serve_dur). Thus, the first and the third conjuncts of the consequent hold. Since Begin_Serve only starts at times that are multiples of 2 * serve_dur and t1 is such a multiple, Begin_Serve cannot start in the interval (t1, t1 + 2 * serve_dur), thus the second conjunct holds.

In the Complete_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
   (   past(EXISTS MSP: microserver
           (MSP.Start(Complete_Serve, t1)), t1)
→ (    ~past(EXISTS MSP: microserver
           (MSP.Start(Begin_Serve, t1)), t1)
   &   FORALL t2: time
              (t1 < t2 & t2 < t1 + serve_dur
       →      ~past(EXISTS MSP: microserver
                        (MSP.Start(Begin_Serve, t2)), t2))
   &   FORALL t2: time
              (t1 < t2 & t2 < t1 + serve_dur
       →      ~past(EXISTS MSP: microserver
                        (MSP.Start(Complete_Serve, t2)), t2))))
```

By previous argument, Begin_Serve can only start at times that are multiples of 2 * serve_dur and Complete_Serve always starts at an end of a Begin_Serve. Since a Complete_Serve starts at t1, Begin_Serve cannot start on any microserver in the interval (t1 - serve_dur, t1 + serve_dur). Thus, the first two conjuncts of the consequent hold. Since Begin_Serve cannot start in the interval (t1 - serve_dur, t1 + serve_dur), Complete_Serve cannot start in the interval (t1, t1 + 2 * serve_dur). Therefore, the third conjunct holds.

VII.8.6.2. Impl_vars_no_change Obligation

For the impl_vars_no_change obligation, it must be shown that the following formula holds. Note that implementation of Vars_No_Change(t1, t1) is not expanded for brevity.

```
FORALL t1, t3: time
   (   t1 ≤ t3
   &   FORALL t2: time
          (   t1 < t2 & t2 ≤ t3
       →      ~past(EXISTS MSP: microserver
                        (MSP.End(Begin_Serve, t2)), t2))
   &   FORALL t2: time
          (   t1 < t2 & t2 ≤ t3
       →      ~past(EXISTS MSP: microserver
                        (MSP.End(Complete_Serve, t2)), t2))
→      FORALL t2: time
          (   t1 ≤ t2 & t2 ≤ t3
       →      IMPL(Vars_No_Change(t1, t2)))))
```

The only way for the implementations of Waiting, Ringing, Busy, Connected_To, and serving_set to change is if the served_set on some microserver changes. The served_set of a microserver only changes when an end of a Begin_Serve or Complete_Serve occurs on that microserver. By the antecedent, there is no such end in the interval (t1, t3], thus the implementations of the variables do not change value in the interval.

VII.9. Parallel Refinement Guidelines

During the specification of the parallel phone system, several issues were encountered that are common to parallel refinement in general. The techniques used to handle these issues in the phone system can be generalized into a set of guidelines for parallel refinement. These guidelines are discussed below.

VII.9.1. Asynchronous Concurrency

Each process in a system performs some set of actions during its execution. In the implementation of a process, it may be neither feasible nor desirable for lower level processes to execute these actions in a lockstep fashion. Instead, lower level processes may need to perform actions dynamically and without synchronization with other lower level processes.

In order to allow such asynchronous concurrency in the refinement of a process, the upper level process needs to be specified appropriately. In particular, concurrent actions in the upper level that may be executed asynchronously in the lower levels should not be specified such that they begin and complete execution in the same transition. For example, in the phone system, the top level could have been specified such that there was a single transition Serve that was executed every t time units in which some set of phones was completely serviced in each execution. This would mean, however, that in the lower levels, phones could only be serviced at the rate of the slowest server and that the servers would process phones in lockstep with eachother.

To allow asynchronous concurrency, concurrent actions in the upper level should be specified such that a set of actions can start and a set of actions can end at every time in the system. For example, in the top level of the central control, the service of a phone was split into the beginning of servicing and the completion of servicing in the transitions Begin_Serve and Complete_Serve. The durations of Begin_Serve and Complete_Serve were set to serve_dur, where 2 * serve_dur was chosen to be a divisor of the duration of every action. In general, it is not necessary to have a separate transition for the beginning and completion of an action. It is necessary, however, to have some record of when an action has started so that it can be completed at the appropriate time. In the central control, changes to the serving variable were used to record this information. When serving changed to true for a phone at time t, that phone began being served at t - serve_dur. Thus, when the duration of the function that was serving

the phone elapsed, the effect of the function was carried out on the phone's state and serving for that phone was reset to false.

VII.9.2. Multiple Writers

In the design of complex systems such as the phone system, there is often a need in the lower levels for multiple processes to control the implementation of a particular upper level variable. In the ASTRAL model, however, only a single process can change the value of a variable, thus it is not possible to let multiple lower level processes change the same variable directly. In the refinement of the central control, the Phone_State variable of the upper level needed to be changed by many of the servers. The solution used in that refinement, which will work in general for any refinement in which multiple writers need to be allowed, was to split the variable into a structure of timestamps, with one timestamp allocated to each process that needs to change the variable.

The Enabled_State type was simple because there were a bounded number of values and each value was the responsibility of a single process. In general, however, the same technique can be used for types with arbitrary values and with an arbitrary number of writers of each value. Consider a variable v of type integer in the upper level. Suppose there are n processes P1 ... Pn that need to change the value of the implementation of v in the lower level to any value. In order to specify this, each process Pi has a variable iv of type "STRUCTURE OF (timestamp: time, value: integer)". Whenever Pi changes the value of iv[value], it sets iv[timestamp] to now.

The mapping for v would then be:

```
IMPL(v) ==
        choose i: integer
            (EXISTS P: Pi
                (   P.iv[value] = i
                &   FORALL P2:proc
                        (P.iv[timestamp] > P2.iv[timestamp])))
```

This states that the value of v is the value of iv[value] of the lower level process that has last changed its iv. Thus, each Pi only changes its own variables and yet the implementation of v can effectively be changed by any Pi.

VII.9.3. Sequential Implementations

In some cases, such as in the central control, there is the possibility that a process may be implemented in both a sequential and a parallel fashion. In these cases, it is necessary for the upper level specification to allow the possibility of multiple actions occurring at the same time and yet not actually requiring multiple actions to occur.

In the top level of the central control, this was achieved by the K_max constant. The K_max restriction in the entry assertion of Begin_Serve limits the number of phones that can be serviced at any given time in

the system. In the sequential refinement, K_max was set to 1, indicating that only one phone at a time can be serviced. In the parallel refinement, K_max was set to the sum of the capacities of the individual servers, indicating that as many phones as is possible for the servers to serve can can be serviced in parallel.

When there is a nondeterministic choice of actions in the upper level, it is necessary to make the choice of actions as a transition parameter in order to allow a sequential refinement of the process. For example, in the top level of the central control, the choice of phones to begin serving was made at the start of Begin_Serve as the set parameter S. This choice could also have been made by a nondeterministic choose expression in the exit assertion of Begin_Serve. This would not have allowed for a sequential refinement of the top level, however, because in the sequential refinement of the central control, as soon as any transition begins execution, the phone and the service to be performed on the phone is immediately known. If the choice of phones and services in the top level was made in the exit assertion, it would not have been possible to determine which phone was going to be served until serve_dur after a phone actually started being served in the top level.

References

[AK 85] Auernheimer, B.; Kemmerer, R.A. "ASLAN user's manual". TRCS84-10, Department of Computer Science, University of California, Santa Barbara, March 1985.

[CGK 97] Coen-Porisini, A.; Ghezzi, C.; Kemmerer, R.A. "Specification of realtime systems using ASTRAL". *IEEE Transactions on Software Engineering*, Sept. 1997, vol. 23, (no. 9): 572-98.

[CKM 95] Coen-Porisini, A.; Kemmerer, R.A.; Mandrioli, D. "A formal framework for ASTRAL inter-level proof obligations". *Proceedings of the 5th European Software Engineering Conference*, Sitges, Spain, 25-28 Sept. 1995. Berlin, Germany: Springer-Verlag, 1995. p. 90-108.

[COR 95] Crow, J.; Owre, S.; Rushby, J.; Shankar, N.; Srivas, M. "A tutorial introduction to PVS". *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, Apr. 1995.

[Kol 99] Kolano, P.Z. "Tools and techniques for the design and systematic analysis of real-time systems". Ph.D. Thesis, University of California, Santa Barbara, 1999.