# Parallel Refinement Mechanisms for Real-Time Systems

Paul Z. Kolano[1], Richard A. Kemmerer[2], and Dino Mandrioli[3]

[1] Trusted Systems Laboratory, Lockheed Martin M & DS – Western Region
3200 Zanker Road, San Jose, CA 95134 U.S.A.
**paul.z.kolano@lmco.com**
[2] Reliable Software Group, Computer Science Department,
University of California, Santa Barbara, CA 93106 U.S.A.
**kemm@cs.ucsb.edu**
[3] Dipartimento di Elettronica e Informazione,
Politecnico di Milano, Milano 20133, Italy
**dino.mandrioli@polimi.it**

**Abstract.** This paper discusses highly general mechanisms for specifying the refinement of a real-time system as a collection of lower level parallel components that preserve the timing and functional requirements of the upper level specification. These mechanisms are discussed in the context of ASTRAL, which is a formal specification language for real-time systems. Refinement is accomplished by mapping all of the elements of an upper level specification into lower level elements that may be split among several parallel components. In addition, actions that can occur in the upper level are mapped to actions of components operating at the lower level. This allows several types of implementation strategies to be specified in a fairly natural way, while the price for generality (in terms of complexity) is paid only when necessary. The refinement mechanisms are illustrated using a simple digital circuit and a much more complex example is sketched.

## 1 Introduction

Refinement is a fundamental design technique that has often challenged the "formal methods" community. In most cases, mathematical elegance and proof manageability have exhibited a deep trade-off with the flexibility and freedom that are often needed in practice to deal with unexpected or critical situations. A typical example is provided by algebraic approaches that exploit some notion of homomorphism between algebraic structures. When applied to parallel systems, such approaches led to the notion of observational equivalence of processes [8] (i.e. the ability of the lower level process to exhibit all and only the observable behaviors of the upper level one). Observational equivalence, however, has been proved too restrictive to deal with general cases and more flexible notions of inter-level relations have been advocated [5].

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. [3] is the origin of the present proposal. [6] addresses the issue within the context of timed Petri nets and the TRIO language. In this approach, a system is modeled as a timed Petri net and its properties are described as TRIO formulas. Then, mechanisms are given that refine the original net into a more detailed one that preserves the original properties. The approach is limited, however, by the expressive power of pure Petri nets, which do not allow one to deal with functional data dependencies. In [11], a system is modeled by an extension of finite

state machines and its properties are expressed in a real-time logic language. Refinement follows a fairly typical algebraic approach by mapping upper level entities into lower level ones and pursuing observational equivalence between the two layers. In this case, observable variables (i.e. variables that are in the process interface), must be identical in the two levels. This leads to a lack of flexibility, as pointed out above, that is even more evident in time dependent systems where refined layers must also guarantee consistency between the occurrence times of the events.

In this paper, we propose highly general refinement mechanisms that allow several types of implementation strategies to be specified in a fairly natural way. In particular, processes can be implemented both sequentially, by refining a single complex transition as a sequence or selection of more elementary transitions, and in a parallel way, by mapping one process into several concurrent ones. This allows one to increase the amount of parallelism through refinement whenever needed or wished.

Also, *asynchronous implementation policies* are allowed in which lower level actions can have durations unrelated to upper level ones, provided that their effects are made visible in the lower level exactly at the times specified by the upper level. For instance, in a phone system, many calls must be served simultaneously, possibly by exploiting concurrent service by many processors. Such services, however, are asynchronous since calls occur in an unpredictable fashion at any instant. Therefore, it is not easy to describe a call service that manages a set of calls within a given time interval in an abstract way that can be naturally refined as a collection of many independent and individual services of single calls, possibly even allowing a dynamic allocation of servers to the phones issuing the calls. In Section 8, we outline how this goal can be achieved by applying the mechanisms described in this paper.

Not surprisingly, generality has a price in terms of complexity. In our approach, however, this price is paid only when necessary. Simple implementation policies yield simple specifications, whereas complex specifications are needed only for sophisticated implementation policies. The same holds for the proof system, which is built hand-in-hand with the implementation mechanisms.

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. Finally, although experience with the application of the proposed mechanisms is still limited, it is not difficult to extract from meaningful examples suitable guidelines for their systematic application to many real cases.

This work is presented in the context of ASTRAL, which is a formal specification language for real-time systems with the following distinguishing features:

- It is rooted in both ASLAN [1], which is an untimed state machine formalism, and TRIO [7], which is a real-time temporal logic, yielding a new, logic-based, process-oriented specification language.
- It has composable modularization mechanisms that allow a complex system to be built as a collection of interacting processes. It also has refinement mechanisms to construct a process as a sequence of layers, where each layer is the implementation of the layer above.
- It has a proof obligation system that allows one to formally prove properties of interest as consequences of process specifications. This proof system is incremental since complex proofs of complex systems can be built by composing small proofs that can be carried out, for the most part, independently of each other. ASTRAL's proofs are of two types. Intra-level proofs guarantee system properties on the basis of local properties that only refer to a single process type. Inter-level proofs guarantee that layer i+1 is a correct implementation of layer i without the need to redo intra-level proofs from scratch.

In this paper, we resume the issue of ASTRAL layering mechanisms and the inter-level proofs, which were addressed in a preliminary and fairly restrictive way in [3].

This paper is structured as follows. Section 2 provides the necessary background on the ASTRAL language. Section 3 summarizes previous purely sequential refinement mechanisms. Section 4 motivates the need for their extensions through a simple running example and illustrates the essentials of the generalized and parallel refinement mechanisms. Section 5 shows their application to the running example. Section 6 presents the proof obligations needed to guarantee implementation correctness and section 7 applies them to the running example. Section 8 briefly summarizes a more complex example. Finally, section 9 provides some concluding remarks. For the sake of conciseness in this paper, we concentrate only on the essentials. Complete technical details can be found in [9].

## 2 ASTRAL Overview

An ASTRAL system specification is comprised of a single global specification and a collection of state machine specifications. Each state machine specification represents a process type of which there may be multiple, statically generated, instances. The *global specification* contains declarations for the process types that comprise the system, types and constants that are shared among more than one process type, and assumptions about the global environment and critical requirements for the whole system.

An ASTRAL *process specification* consists of a sequence of *levels*. Each level is an abstract data type view of the process being specified. The first ("top level") view is a very abstract model of what constitutes the process (types, constants, variables), what the process does (state transitions), and the critical requirements the process must meet (invariants and schedules). Lower levels are increasingly more detailed with the lowest level corresponding closely to high level code.

The process being specified is thought of as being in various *states*, where one state is differentiated from another by the values of the *state variables*, which can be changed only by means of *state transitions*. Transitions are specified in terms of entry and exit assertions, where *entry assertions* describe the constraints that state variables must satisfy in order for the transition to fire, and *exit assertions* describe the constraints that are fulfilled by state variables after the transition has fired. An explicit non-null duration is associated with each transition. Transitions are executed as soon as they are enabled if no other transition is executing in that process.

Every process can export both state variables and transitions. Exported variables are readable by other processes while exported transitions are callable from the external environment. Inter-process communication is accomplished by inquiring about the values of exported variables and the start and end times of exported transitions.

In addition to specifying system state (through process variables and constants) and system evolution (through transitions), an ASTRAL specification also defines system critical requirements and assumptions about the behavior of the environment that interacts with the system. Assumptions about the behavior of the environment are expressed by means of *environment clauses* that describe the pattern of calls to external transitions, which are the stimuli to which the system reacts. Critical requirements are expressed by means of *invariants* and *schedules*. Invariants represent requirements that must hold in every state reachable from the initial state, no matter what the behavior of the external environment is, while schedules represent

additional properties that must be satisfied provided that the external environment behaves as assumed.

Invariants and schedules are proved over all possible executions of a system. A system execution is a set of process executions that contains one process execution for each process instance in the system. A process execution for a given process instance is a history of events on that instance. The value of an expression E at a time t1 in the history can be obtained using the *past* operator, past(E, t1). There are four types of events in ASTRAL. A *call event*, Call(tr1, t1), occurs for an exported transition tr1 at a time t1 iff tr1 was called from the external environment at t1. A *start event*, Start(tr1, t1), occurs for a transition tr1 at a time t1 iff tr1 fires at t1. Similarly, an *end event*, End(tr1, t1), occurs if tr1 ends at t1. Finally, a *change event*, Change(v1, t1), occurs for a variable v1 at a time t1 iff v1 changes value at t1. Note that change events can only occur when an end event occurs for some transition. An introduction and complete overview of the ASTRAL language can be found in [2].

The example system used throughout the remainder of the paper is shown in figure 1. 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 Mult_Add                        AXIOM     TRUE
EXPORT       compute, output            INVARIANT
CONSTANT     dur1: pos_real               FORALL t1: time, a, b, c, d: integer
VARIABLE     output: integer                ( Start(compute(a, b, c, d), t1)
TRANSITION compute(a,b,c,d: integer)      →  FORALL t2: time
  ENTRY      [TIME: dur1]                        ( t1 + dur1 ≤ t2 & t2 ≤ now
    TRUE                                       →  past(output, t2) = a * b + c * d))
  EXIT   output = a * b + c * d
```

## 3 Sequential Refinement Mechanism

A refinement mechanism for ASTRAL was defined in [3]. 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 $P_U$ and $P_L$, where $P_L$ is a refinement of $P_U$, the implementation statement, hereafter referred to as the IMPL mapping, defines a mapping from all the types, constants, variables, and transitions of $P_U$ into their corresponding terms in $P_L$, which are referred to as *mapped* types, constants, variables, or transitions. $P_L$ can also introduce types, constants and/or variables that are not mapped, which are referred to as the *new* types, constants, or variables of $P_L$. Note that $P_L$ cannot introduce any new transitions (i.e. each transition of $P_L$ must be a mapped transition). A transition of $P_U$ can be mapped into a sequence of transitions, a selection of transitions, or any combinations thereof.

A selection mapping of the form $T_U == A_1 \ \& \ T_{L.1} \mid A_2 \ \& \ T_{L.2} \mid ... \mid A_n \ \& \ T_{L.n}$, is defined such that when the upper level transition $T_U$ fires, one and only one lower level transition $T_{L.j}$ fires, where $T_{L.j}$ can only fire when both its entry assertion and its associated "guard" $A_j$ are true.

A sequence mapping of the form $T_U$ == WHEN $Entry_L$ DO $T_{L.1}$ BEFORE $T_{L.2}$ BEFORE ... BEFORE $T_{L.n}$ OD, defines a mapping such that the sequence of transitions $T_{L.1}$; ...; $T_{L.n}$ is enabled (i.e. can start) whenever $Entry_L$ 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 $T_U$ corresponds to the starting time of the sequence (which is not necessarily equal to the starting time of $T_{L.1}$ because of a possible delay between the time when the sequence starts and the

time when $T_{L.1}$ becomes enabled), while the ending time of $T_U$ corresponds to the ending time of the last transition in the sequence, $T_{L.n}$. 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 $T_U$ are broadcast when $T_U$ terminates. Thus, mapped variables of $P_L$ can be modified only when the sequence implementing $T_U$ ends.

The inter-level proofs consist of showing that 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 $T_U$ fires, one of the lower level transitions $T_{L.j}$ fires, that the effect (i.e. changes to variables) of each $T_{L.j}$ is equivalent to the effect of $T_U$, and that the duration of each $T_{L.j}$ is equal to the duration of $T_U$. For sequences, it must be shown that the sequence is enabled iff $T_U$ is enabled, that the effect of the sequence is equivalent to the effect of $T_U$, and that the duration of the sequence (including any initial delay after $Entry_L$ is true) is equal to the duration of $T_U$.

## 4 Parallel Refinement Mechanism

In the sequential mechanism, refinement occurs at the transition level, where the behavior of each upper level transition can be specified in greater detail at the lower level. We now extend the ASTRAL refinement mechanism to include process level refinement, which allows a process to be refined as a collection of components that operate in parallel. For example, a reasonable refinement of the Mult_Add circuit is shown in figure 2. Here, the refinement of the system consists of two multipliers that compute a * b and c * d in parallel and an adder that adds the products together and produces the sum. This refinement cannot be expressed in the sequential mechanism due to the parallelism between the two multipliers. The new parallel mechanism introduced below, however, easily expresses this refinement.


**Fig. 1.** Mult_Add circuit

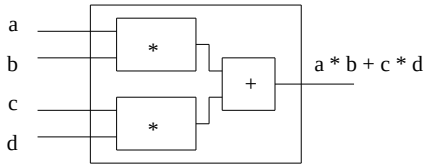


**Fig. 2.** Refined Mult_Add circuit

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 the events that occur in the lower level when the transition is executed in the upper level. It must then be shown that these events will only occur when the upper level transition ends and that the effect will be equivalent. Like the sequential refinement mechanism of [3], an IMPL mapping is used, which describes how items in an upper level are implemented by items in the next lower level. The items of the upper level include variables, constants, types, and transitions. In addition, the implementation mapping must describe how upper level expressions are transformed into lower level expressions. The following sections only discuss the transition mappings. A complete description of the IMPL mapping is given in [9].

### 4.1 Parallel Sequences and Selections

A natural but limited approach to defining parallel transition mappings is to extend the sequential sequence and selection mappings into parallel sequence and selection mappings. Thus, a "||" operator could be allowed in transition mappings, such that "$P_1$.tr1 || $P_2$.tr2" indicates that tr1 and tr2 occur in parallel on processes $P_1$ and $P_2$, 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(M1.product, M2.product) OD,
```

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 produce transition 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 $P_U$ and the lower level production cells $P_{L.1}$ and $P_{L.2}$ are shown in figure 3. Note that the first transition executed on $P_U$ is init, which represents the "warm-up" time of the production cell in which no output is produced.

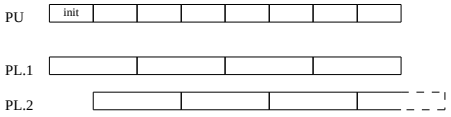


**Fig. 3.** Production cell refinement

This refinement cannot be expressed using parallel sequences and selections because there is no sequence of parallel transitions in the lower level that corresponds directly to produce in 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.

### 4.2 Parallel Start, End, and Call Mappings

The desired degree of flexibility is obtained by using transition mappings that are based on the start, end, and call of each transition. For each upper level transition $T_U$, a start mapping "IMPL(Start($T_U$, now)) == $Start_L$" and an end mapping "IMPL(End($T_U$, now)) == $End_L$" must be defined. If $T_U$ is exported, a call mapping "IMPL(Call($T_U$, now)) == $Call_L$" must also be defined. These mappings are defined at time now, which is a special global variable that holds the current time in the system; thus, the mappings are defined for all possible times.

Here, $Start_L$, $End_L$, and $Call_L$ are well-formed formulas using lower level transitions and variables. For the most part, the end and call mappings will correspond to the end or 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, the current time, etc. Call mappings are restricted such that for every lower level exported transition $T_L$, Call($T_L$) must be referenced in some upper

level exported transition call mapping IMPL(Call($T_U$, 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. Any parameter referenced in a call mapping must be mapped to a call parameter of some lower level transition and the corresponding start mapping must contain the same transitions as the call mapping. Thus, the start and end parameters are taken from the associated set of unserviced call parameters.

With these mappings, the initialize and produce transitions can be mapped as follows. Note that the parallelism between $P_{L.1}$ and $P_{L.2}$ is implied by the overlapping start and end times of produce on each process and not by a built-in parallel operator.

```
IMPL(Start(initialize, now)) ==
    now = 0 & P_L.1.Start(produce, now)
IMPL(Start(produce, now)) ==
    IF now mod 2 = 0
    THEN P_L.1.Start(produce, now)
    ELSE P_L.2.Start(produce, now)
    FI
```

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

## 5 The Mult_Add Circuit

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

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

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

The lower level consists of two instances of the Multiplier process type, M1 and M2, and one instance of the Adder process type, A1. The output variable of the upper level 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, multiply starts on both M1 and M2. 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)
```

## 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.

### 6.1 Proof Obligations

Instead of proving directly that the mappings hold at all times, it can be shown that the mappings hold indirectly by proving that they preserve the axiomatization of the ASTRAL abstract machine, thus they preserve any reasoning performed in the upper level. This can be accomplished by proving the implementation of each abstract machine axiom. The proof obligations are shown in figure 4 and are written in the specification language of PVS [4], which will be used to assist the proofs in the future.

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: transition_ll, t1: time):
        Exported(tr_ll) AND Call(tr_ll, t1)(t1) IMPLIES
            (EXISTS (tr_ul: transition_ul): (FORALL (t2: time):
                IMPL(Call(tr_ul, t2)(t2)) IMPLIES Call(tr_ll, t2)(t2)) 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), now)) == M1.Call(multiply(a, b), now) & M2.Call(multiply(c, d), now)". 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.

An assumption is also needed to assure that whenever the parameters of an upper level exported transition are distributed among multiple transitions at the lower level, the collection of parameters for which the lower level transitions execute come from a single set of call parameters. For example, in the Mult_Add circuit, the upper level compute transition may be called with two sets of parameters {1, 2, 3, 4} and {5, 6, 7, 8} at the same instant. In the lower level implementation, the multiply transition of each multiplier takes two of the parameters from each upper level call. Thus, in the example, multiply is enabled on M1 for {1, 2} and {5, 6} and on M2 for {3, 4} and {7, 8}. Without an appropriate assumption, M1 may choose {1, 2} and M2 may choose {7, 8}, thus computing the product for {1, 2, 7, 8}, which was not requested at the upper level. The *impl_call_fire_parms* assumption given in [9] prevents this.

```
impl_end1: OBLIGATION
  (FORALL (tr1: transition, t1: time):
    IMPL(End(tr1, t1)(t1)) IMPLIES
      t1 ≥ IMPL(Duration(tr1)))
impl_end2: OBLIGATION
  (FORALL (tr1: transition, t1: time, t2: time):
    t1 = t2 - IMPL(Duration(tr1)) IMPLIES
      (IMPL(Start(tr1, t1)(t1)) IFF
        IMPL(End(tr1, t2)(t2))))
impl_trans_mutex: OBLIGATION
  (FORALL (tr1: transition, t1: time):
    IMPL(Start(tr1, t1)(t1)) IMPLIES
      (FORALL (tr2: transition):
        tr2 ≠ tr1 IMPLIES
          NOT IMPL(Start(tr2, t1)(t1))) AND
      (FORALL (tr2: transition, t2: time):
        t1 < t2 AND
        t2 < t1 + IMPL(Duration(tr1)) IMPLIES
          NOT IMPL(Start(tr2, t2)(t2))))
impl_trans_entry: OBLIGATION
  (FORALL (tr1: transition, t1: time):
    IMPL(Start(tr1, t1)(t1)) IMPLIES
      IMPL(Entry(tr1, t1)))
impl_trans_exit: OBLIGATION
  (FORALL (tr1: transition, t1: time):
    IMPL(End(tr1, t1)(t1)) IMPLIES
      IMPL(Exit(tr1, t1)))
```

```
impl_trans_called: OBLIGATION
  (FORALL (tr1: transition, t1: time):
    IMPL(Start(tr1, t1)(t1)) AND
    Exported(tr1) IMPLIES
      IMPL(Issued_Call(tr1, t1)))
impl_trans_fire: OBLIGATION
  (FORALL (t1: time):
    (EXISTS (tr1: transition):
      IMPL(Enabled(tr1, t1))) AND
    (FORALL (tr2: transition, t2: time):
      t1 - IMPL(Duration(tr2)) < t2 AND
      t2 < t1 IMPLIES
        NOT IMPL(Start(tr2, t2)(t2))) IMPLIES
    (EXISTS (tr1: transition):
      IMPL(Start(tr1, t1)(t1))))
impl_vars_no_change: OBLIGATION
  (FORALL (t1: time, t3: time):
    t1 ≤ t3 AND
    (FORALL (tr2: transition, t2: time):
      t1 < t2 AND t2 ≤ t3 IMPLIES
        NOT IMPL(End(tr2, t2)(t2))) IMPLIES
      (FORALL (t2: time):
        t1 ≤ t2 AND t2 ≤ t3 IMPLIES
          IMPL(Vars_No_Change(t1, t2))))
impl_initial_state: OBLIGATION
  IMPL(Initial(0))
impl_local_axiom: OBLIGATION
  (FORALL (t1): IMPL(Axiom(t1)))
```

**Fig. 4.** Parallel refinement proof obligations

In the axiomatization of the ASTRAL abstract machine [9], 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 a start of tr1 occurred at t1 and an end of tr1 occurred at t1 + Duration(tr1). Additionally, since an end of tr1 can only occur at t1 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. 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.

The other obligations are the mappings of the ASTRAL abstract machine axioms. 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. The *impl_trans_entry* obligation ensures that any time the mapped start of a transition occurs, the mapped entry assertion of the transition holds. The *impl_trans_exit* obligation ensures that any time the mapped end of a transition occurs, the mapped exit assertion of the transition holds. 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. 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. The *impl_vars_no_change* obligation ensures that mapped variables only change value when the mapped end of a transition occurs. The *impl_initial_state* obligation ensures that the mapped initial clause holds at time zero.

Besides the abstract machine axioms, the local proofs of ASTRAL process specifications can also reference the local axiom clause of the process (which is empty (i.e. TRUE) in the Mult_Add circuit). 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.

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

### 6.2 Correctness of Proof Obligations

The proof obligations for the parallel refinement mechanism given in figure 4 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, $I \vdash_{f1/a1} I_1 \vdash_{f2/a2} \ldots \vdash_{fn/an}$ TRUE, where each transformation $f_i/a_i$ corresponds to the application of a series $f_i$ of first-order logic axioms and a single abstract machine axiom $a_i$. 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($I_1$) $\vdash_{f2'/impl\_a2}$ ... $\vdash_{fn'/impl\_an}$ TRUE can be constructed by replacing the application of each abstract machine axiom $a_i$ by impl_$a_i$. Additionally, each series $f_i$ of first-order logic axioms is replaced by a series $f_i'$ that takes any changes to the types of variables and constants into consideration.

## 7 Proof of Mult_Add Circuit Refinement

This section shows the most notable cases of the parallel refinement proof obligations for the Mult_Add circuit. The full proofs can be found in [9]. The obligations below were obtained from the corresponding obligations in figure 4 by expanding the IMPL mapping appropriately, replacing quantification over transitions with the actual transitions of the Mult_Add circuit, rewriting the Curried PVS form (e.g. Start(tr1, t1)(t2)) to its ASTRAL equivalent (e.g. past(Start(tr1, t1), t2)), and performing some minor simplifications. For the impl_end2 obligation, the following must be proved.

```
FORALL t1: time
    (   past(M1.Start(multiply, t1 - 3), t1 - 3)
    &   past(M2.Start(multiply, t1 - 3), t1 - 3)
↔       past(A1.End(add, t1), t1))
```

For the forward direction, it must be shown that add ends on A1 at t1, thus starts 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. The reverse direction is similar.

For the impl_trans_exit obligation, the formula below must be proved.

```
FORALL t1: time
    (   past(A1.End(add, t1), t1)
→       FORALL a, b, c, d: integer
            (   past(M1.Start(multiply(a, b), t1 - 3), t1 - 3)
            &   past(M2.Start(multiply(c, d), t1 - 3), t1 - 3)
        →       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. Since multiply ends on M1 and M2 at t1 - 1, it starts on M1 and M2 at t1 - 3 for two pairs of parameters (a, b) and (c, d), respectively, which were provided by the external environment. 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.

The impl_trans_fire obligation is given below.

```
FORALL t1: time
    (   EXISTS t2: time
            (   t2 ≤ t1
            &   past(M1.Call(multiply, t2), t1)
            &   past(M2.Call(multiply, t2), t1)
            &   FORALL t3: time
                    (   t2 ≤ t3 & t3 < t1
                →   ~   (   past(M1.Start(multiply, t3), t3)
```

```
                    &   past(M2.Start(multiply, t3), t3))))
    &   FORALL t2: time
            (   t1 - 3 < t2 & t2 < t1
        →   ~   (   past(M1.Start(multiply, t2), t2)
                &   past(M2.Start(multiply, t2), t2)))
→       past(M1.Start(multiply, t1), t1)
    &   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 zero, 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. At time zero, multiply cannot have ended, thus the entry assertion of multiply is true, so if both are called, both fire. 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 some time T0. Suppose multiply occurs on M1 (the M2 case is similar), then M1 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 cannot be executing multiply at T0 or else M1 must also be executing multiply by the inductive hypothesis, thus M2 must be idle. Similarly, it must have been at least one time unit since multiply ended on M2. By impl_call, multiply must have been called on M2 since it was called on M1. Thus, multiply is enabled on M2, so must fire. Therefore, M1.Start(multiply) and M2.Start(multiply) always occur at the same time. Based on this fact, the following two expressions are equivalent.

```
FORALL t3: time                                 FORALL t3: time
  ( t2 ≤ t3 & t3 < t1                             ( t2 ≤ t3 & t3 < t1
→ ~ ( past(M1.Start(multiply, t3), t3)          →  ~past(M1.Start(multiply, t3), t3)
    & past(M2.Start(multiply, t3), t3)))            & ~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 at t1. Since the entry of multiply holds, multiply has been called but not yet serviced, and M1 is idle, multiply starts on M1 at t1. Since multiply always starts on both M1 and M2 at the same time as shown above, impl_trans_fire holds.

The remaining proof obligations were all proved in a straightforward manner. Therefore, the lower level is a correct refinement of the upper level and 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)
→       FORALL t2: time
            (   t1 + dur1 ≤ t2 & t2 ≤ now
        →       past(A1.sum, t2) = a * b + c * d))
```

## 8 Parallel Phone System

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. Furthermore, the proof obligations for a simple implementation were themselves simple. In this section we briefly outline how our mechanisms can be

applied to the specification and refinement of a much more complex case such as the control of a phone system.

The system considered here is a slightly modified version of the phone system defined in [2]. It 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. Due to space limitations, this example cannot be described in full detail. Thus, we limit ourselves to an informal description of the main steps of the refinement process and refer the interested reader to [9] for the complete description and proof.

The specification of the central control, which is the core of the whole system, is articulated into three layers. The goal of the top level is to provide an abstract and global view of the supplied services in such a way that the user can have complete and precise knowledge of the external behavior, both in terms of functions performed and in terms of service times, of the central control but the designer still has total freedom on implementation policies. In fact, as a result, the description provided in [2] is just an alternative implementation of the top level description given here, which differs from this version in that services are granted sequentially rather than in parallel.

To achieve our goal (i.e. to allow the implementation of services both asynchronously in parallel and strictly sequentially, as suggested by figures 5 and 6), the top level is specified such that a *set* of services can start and a set of services can end at every time unit in the system (for simplicity we assume discrete time). In these figures, Ti_si.Pk denotes providing service si to phone k.

The service of a phone is split into the beginning of servicing and the completion of servicing through two transitions: Begin_Serve and Complete_Serve. In other words, instead of assigning ASTRAL transitions to single services or groups thereof, we only make the beginning and the end of services visible at the top level. In this way, we do not commit too early to a fixed duration of the implementation of the service, stating only when a service will begin and when it will be completed. Thus, the durations of Begin_Serve and Complete_Serve are set to "serve_dur", where 2*serve_dur is chosen to be a divisor of the duration of every service.

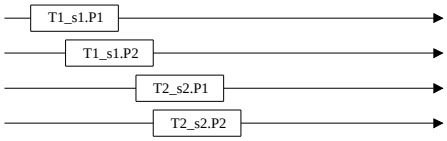


**Fig. 5.** Parallel implementation of different services on several phones

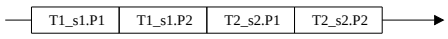


**Fig. 6.** Sequential implementation of different services on several phones

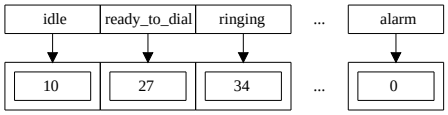


**Fig. 7.** Implementation of the phone state

A parameterized variable "serving(P)" records when each phone P began being serviced so that it can complete being serviced at the appropriate time. When serving(P) changes to true for a phone P at time t, P began being served at t - serve_dur. Thus, when the duration of the function that was serving the phone elapses from this time, Complete_Serve carries out the effect of the function on the phone's state and resets serving for that phone to false.

To allow both a sequential and a parallel implementation, it is necessary for the top level specification to allow the possibility of multiple actions occurring at the same time without actually requiring multiple actions to occur. This is achieved by limiting the number of phones that can be serviced at any given time to be less than a

constant “K_max”. In the sequential refinement, K_max is mapped to one, indicating that only one phone at a time can be serviced. In the parallel refinement, K_max is mapped to the sum of the capacities of the individual servers, indicating that as many phones as it is possible for the servers to serve can be serviced in parallel.

Let us now look at the parallel implementation of the top level central control. As mentioned earlier, this is achieved through two refined layers. In the first refinement, the central control is split into several parallel processes, each of which is devoted to a single service of the top level central control. Thus, there is a process devoted to giving dial tone, a process devoted to processing entered digits, and so on. 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. That is, the “Phone_State(P)” variable in the top level, which holds the state of each phone P (Phone_State can take the values idle, ready_to_dial, ringing, ...), needs to be split among all the servers in the lower level.

In the ASTRAL model, however, only a single process can change the value of a variable, thus it is not possible to let all of the lower level servers change the same variable directly. A solution to this problem is to split Phone_State into a set of “timestamp” variables, with one timestamp for each possible state of a phone. Each server is then allocated the timestamp variables associated with the phone state(s) it is responsible for. The state of a phone is the state associated with the timestamp that has last changed among all the servers. Figure 7 illustrates such a state variable mapping. In this figure, each component is managed by a different process and the current state of the phone is “ringing” because the corresponding timestamp component holds the maximum value.

Finally, in the third layer of the central control, each second level server is refined by a parallel array of “microservers”, where each microserver is devoted to processing the calls of a single phone. Each microserver picks a phone from the set of phones waiting for the service 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 these sets over all the microservers implements the set of phones that are being served on the upper level server.

This example demonstrates that the parallel refinement mechanism can be used to express very complex parallel implementations. Not surprisingly, such generality is obtained at the cost of complicating the proofs of the proof obligations as shown in the major proofs of the phone system refinement, which were completed in [9]. We are confident that the approach adopted in this and other examples can be applied in a fairly similar way to a collection of real-life cases. As a result, we should obtain a general and systematic *method* for driving the specification and refinement of parallel and asynchronous systems.

## 9 Conclusions, Future Work, and Acknowledgments

ASTRAL aims to provide a disciplined and well-structured way of developing real-time systems. To achieve this goal it stresses modularization and incremental development through several refinement levels. In this paper, we presented an approach to accomplish these refinement steps in a flexible, general, and yet rigorous and provably correct way. A key feature of the proposed mechanism is the exploitation of parallelism so that global actions at a high level of abstraction may be described as individual transitions that can be refined at a lower level as several

concurrent and cooperating activities. Our approach allows more generality and flexibility than the few independent ones available in the literature, which are more algebraic in nature (for an up-to-date survey of formal refinement methods see [5]).

Although our experience in the application of the mechanisms to problems of practical interest is still limited, we have already developed a number of case studies that show that the approach naturally scales up. Besides those summarized in this paper, we also mention the management of a hydroelectric reservoir system based on a real-life project. Early methods guiding the user throughout the development of real-life cases have already been extracted from these experiences [9].

As usual, when moving towards complex applications, the support of suitable tools is fundamental. ASTRAL is already equipped with several prototype tools [10] that allow the user to edit and manage complex specifications as well as providing support for their analysis. In particular, proof obligations can be automatically produced from specifications, and proofs are supported by both model checking and deductive facilities. The model checker can check the critical requirements of a particular instance of a specification over a finite time interval. ASTRAL has been encoded into the language of the PVS theorem prover to support deductive reasoning.

The ASTRAL toolset currently supports the sequential refinement mechanism of [3], but does not yet support the parallel refinement mechanism of this paper. To support the new mechanism, the design portion needs to incorporate a slightly different specification structure, an algorithm is needed to transform upper level expressions to lower level expressions using the implementation mapping, and the proof obligations must be incorporated into the current ASTRAL-PVS library for use with the theorem prover. It may be possible to use the PVS rewriting mechanism directly to transform upper level items to lower level items. This work is currently under way. In addition to tool support, more parallel refinements will be developed to test the expressiveness and applicability of the new parallel mechanism.

The authors would like to thank Klaus-Peter Loehr for his participation in the development of the parallel mechanism and Giovanna Di Marzo for her valuable comments. This research was partially supported by NSF Grant No. CCR-9204249 and by the Programma di scambi internazionale of the CNR.

## References

1. Auernheimer, B. and R.A. Kemmerer. ASLAN User's Manual. Technical Report TRCS84-10, Department of Computer Science, University of California, Santa Barbara, Mar. 1985.
2. Coen-Porisini, A., C. Ghezzi, and R.A. Kemmerer. "Specification of Realtime Systems Using ASTRAL". *IEEE Transactions on Software Engineering*, Sept. 1997, vol. 23, (no. 9): 572-98.
3. Coen-Porisini, A., R.A. Kemmerer, and D. Mandrioli. "A Formal Framework for ASTRAL Inter-level Proof Obligations". *Proc. of the 5th European Software Engineering Conf.*, Sitges, Spain, Sept. 1995.
4. Crow, J., S. Owre, J. Rushby, N. Shankar, and M. Srivas. "A Tutorial Introduction to PVS". *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, Apr. 1995.
5. Di Marzo Serugendo, G. "Stepwise Refinement of Formal Specifications Based on Logical Formulae: From CO-OPN/2 Specifications to Java Programs". Ph.D. Thesis no. 1931, EPFL, Lausanne, 1999.
6. Felder M., A. Gargantini, and A. Morzenti. "A Theory of Implementation and Refinement in Timed Petri Nets". *Theoretical Computer Science*, July 1998, vol. 202, (no. 1-2): 127-61.
7. Ghezzi, C., D. Mandrioli, and A. Morzenti. "TRIO: a Logic Language for Executable Specifications of Real-time Systems". *Jour. of Systems and Software*, May 1990, vol. 12, (no. 2): 107-23.
8. Hennessy, A. and R. Milner. "Algebraic Laws for Nondeterminism and Concurrency". *Jour. of the ACM*, Jan. 1985, vol. 32, (no. 1): 137-61.
9. Kolano, P.Z. "Tools and Techniques for the Design and Systematic Analysis of Real-Time Systems". Ph.D. Thesis, University of California, Santa Barbara, Dec. 1999.
10. Kolano, P.Z., Z. Dang, and R.A. Kemmerer. "The Design and Analysis of Real-time Systems Using the ASTRAL Software Development Environment". *Annals of Software Engineering*, vol. 7, 1999.
11. Ostroff, J.S. "Composition and Refinement of Discrete Real-Time Systems". *ACM Transactions on Software Engineering and Methodology*, Jan. 1999, vol. 8, (no. 1): 1-48.