Ordering strict partial orders to model behavioral refinement

Mathieu Montin
Université de Toulouse ; Toulouse INP, *IRIT*
2 rue Camichel, BP 7122, 31071 Toulouse Cedex 7, France
CNRS ; Institut de Recherche en Informatique de Toulouse (IRIT)
mathieu.montin@enseeiht.fr

Marc Pantel
Université de Toulouse ; Toulouse INP, *IRIT*
2 rue Camichel, BP 7122, 31071 Toulouse Cedex 7, France
CNRS ; Institut de Recherche en Informatique de Toulouse (IRIT)
marc.pantel@enseeiht.fr

Software is now ubiquitous and involved in complex interactions with the human users and the physical world in so-called cyber-physical systems (CPS) where the management of time is a major issue. Separation of concerns is a key asset in the development of these ever more complex systems. Two different kinds of separation exist: a first one corresponds to the different steps in a development leading from the abstract requirements to the system implementation and is qualified as vertical. It matches the commonly used notion of refinement. A second one corresponds to the various components in the system architecture at a given level of refinement and is called horizontal. Refinement has been studied thoroughly for the data, functional and concurrency concerns while our work focuses on the time modeling concern. This contribution aims at providing a formal construct for the verification of refinement in time models, through the definition of an order between strict partial orders used to relate the different instants in asynchronous systems. This relation allows the designer at the concrete level to distinguish events that are coincident at the abstract level while preserving the properties assessed at the abstract level. This work has been conducted using the proof assistant Agda and is connected to a previous work on the asynchronous language CCSL, which has also been modelled using the same tool.

1 Introduction

1.1 Separation of concerns

Nowadays, many devices require to handle complex interactions with both the human users and the physical world. These devices, like cars, aircrafts, trains, rockets, satellites, pacemakers, robots, etc are called Cyber-Physical systems (CPS). While these ones offer more and more advanced and complex services, they become increasingly dense and complex, which leads their developers to use separation of concerns throughout the different phases of their development. There exists two kinds of separations of concerns : the first one is qualified as horizontal and aims at describing complex systems through the different physical – or logical – parts they contain. The second one is qualified as vertical and corresponds to the commonly used notion of refinement, where the different levels of abstraction of a given systems are described separately while the properties they exhibit are preserved throughout these steps.

Horizontal separation is usually handled at design time through the expression of the various system parts in different Domain Specific Modelling Languages (DSML). Their execution, for validation and

Submitted to:
REFINE 2018

verification purposes, may rely on different Models of Computation (MoC). A sophisticated coordination of the various events occurring in the different parts is thus needed to observe the global behavior of the system. For CPS, the modeling of time in the various DSML and the coordination between the different time models is a major issue. This heterogenous modelling approach has been integrated in the Ptolemy toolset proposed by Lee et al. [11], the ModHel'X toolset proposed by Boulanger et al. [20] and the GEMOC studio proposed by Combemale et al. [12]. Our work targets a proof based formal modeling and verification framework to prove properties of languages and models in such toolsets.

Vertical separation usually enforces a refinement relation between the different models of the same part of the system in order to ensure the consistency of the various global executions. This approach is for example advocated by the B and Event-B methods [2, 3, 4] in order to prove the preservation of the properties from the specification to the implementation. In the case of asynchronous systems, refinement is usually related to simulation and corresponds to replacing τ transition by effective actions. In the case of synchronous systems, refinement corresponds to decomposing an instant at a given level into several instants at the refined level. Synchronous refinement has been widely studied in the case of synchronous MoC first as oversampling for data-flow languages [32] and then as time refinement for reactive languages [18, 29]. Polychronous time models have been used to assess the vertical refinement during system design [39]. Their relational nature is more appropriate at design time as it introduces less constraints than the common functional computation of clocks in synchronous programming languages derived from LUSTRE. Thus, the refinement has to be made explicit in our formal framework.

1.2 Context

Our work focuses on the modeling of time in GEMOC that mixes both horizontal and vertical separation of concerns. Indeed, GEMOC allows to define the DSML used to model the various parts in a CPS in each phase of their development. Thus, DSML are combined both in an horizontal and vertical manners. GEMOC relies on the UML MARTE CCSL (Clock Constraint Specific Language) to model both the MoC for the various DSML [13, 15, 25] and the coordination between DSML using the Behavioral Coordination Language (BECOOL) [24]. Our work in GEMOC targeted an example of horizontal separation. This contribution targets the vertical separation. More precisely, we want to assess the relations between the various time concerns in the various models of the same system part in a vertical separation of concerns. In that purpose, we provide a mechanized definition for the vertical relation and eventually apply it to CCSL.

This issue is handled by introducing an instant refinement relation inspired from time refinement in order to ultimately combine both horizontal and vertical separation of concern in the design of heterogeneous systems. In time models that depict the temporal execution of heterogeneous systems, partial orders are usually used to bind the instants together. This contribution provides a formal construct for the time refinement in these models, as an order relation between these partial orders. This relation allows the designer at the concrete level to distinguish events that are coincident at the abstract level while preserving the properties assessed at the abstract level.

This relation is generic and can be applied to any system, the semantics of which relies on a set of traces. It has been mechanized with the Agda proof assistant, in order to be linked with a denotational semantics of CCSL, that has already been mechanized using Agda. This allows assessing properties of this new relation and prove that it preserves the different CCSL operators semantics. This contribution relies on a simple example of oversampling in a synchronous system.

1.3 State of the art

Refinement has been thoroughly studied [36, 37] and implemented for many different modeling and programming concerns like data [38] and algorithms (sequential [7], concurrent [6], distributed, etc). Time can be represented with a single global reference clock that binds all clocks in the system together [27, 10]. However, since building these global clocks is usually tricky, time is more often abstracted as a partial order relation [35, 26]. Refinement [1] then relies on simulation [21, 22] or bisimulation relations between the semantics of the more abstract and concrete system models.

Our proposal provides a mechanized refinement relation formalized in the Agda proof assistant. We target its coupling with a previous mechanization of the semantics of CCSL in the same proof assistant. This relation can be integrated with any other concurrent languages. Formal mechanization of time models has already been done using other formal methods, for example [19] uses Higher Order Logic in Isabelle/HOL; [17] and [34] use the Calculus of Inductive Constructions in Coq, see [8]. The use of Agda in this development is motivated by the expressiveness of the language and its underlying unification mechanism, which provides an efficient interactive proof experience that other tools might lack. More on Agda can be found in [33], [28] and [9]. Although Agda differs from Coq by several aspects, both of these tools rely on the same underlying intuitionist type theory, first described in [30] and clarified in [31]. The paper version of CCSL denotational semantics, which is connected to this work, can be found in [14]. TimeSquare, the tool developed to describe CCSL systems as well as solve constraint sets has been presented in [16]. As for CCSL itself, it was first presented in [5].

1.4 Two different ways of considering refinement

Refinement is a relation between the trace semantics of two systems, a more abstract and a more concrete one that can be assessed in two different ways. While these possibilities rely on different approaches, they are ultimately equivalent. Either the concrete system is derived from the first system in a correct by construction manner, and refinement is ensured by the a correct by construction derivation method, or both systems are provided independently and refinement is assessed relying on mapping information between both systems.

The first approach can be considered as an accretion of events. It is advocated for example by the Event-B method. It consists in building step by step correct by construction trace generators (through an operational semantics). Each step consisting in a layer of concretization, hence a layer of refinement. At a given time in this process, only the events used in the levels already described are existing. This means that the set of events evolves throughout the development of the system. This operational view is akin, as explained, to correct-by-construction development of systems from the abstract specification to the concrete executable program. Sets of traces can then possibly be created through the different possible executions of the more concrete version of the system. The relations binding the different events in these traces are deduced from properties written on the specification and preserved through the refinement process.

The second approach relies on building both sets of possible traces itself. Refinement can be assessed on these sets (usually described through a partial order over the instants on which events occur) regardless of the generators of these traces. This vision require to express refinement on the traces themselves, through relations instead of functions. This approach can be applied later in the development process as it does not require the system to be built throughout a correct-by-construction methodology. The traces contain all the events of the system, regardless of the level of refinement on which they appear. This paper present a relation that has to be satisfied in order to verify refinement in such cases. This relation,

rather than constraining the generator of the traces or the events themselves, constrain the partial order that bind them in each layer of refinement.

These two visions are somewhat conceptually opposed and the tools used to model and describe them differ as well. We chose to use Agda in this work. Set theory is akin to describe the second approach as it naturally embeds the operation of accretion through its axioms. In type theories, as the one on which Agda is based, subsets and union are not natural, while these tools provide the right level of expressiveness to mechanize relations between quantities. This motivates the use of Agda for this work where we remain descriptive and never actually compute the traces of events on which our relation is defined.

2 Time and refinement

This section briefly introduces notions inherent to time handling in asynchronous systems, from the instants to the strict partial orders binding them in time models, then proceeds to the core of our contribution: our relation of refinement between these orders.

2.1 Instants

Instants are the main concept on which concurrent languages are defined. Informally, an instant is a point in time where events can occur. It matches, to a certain extend, the common vision one has about time. However, time in asynchronous systems cannot be easily depicted as a single time-line consisting of well ordered instants. This is due to the lack of knowledge one can have regarding the execution of such systems, when it is usually impossible to know, for all events and their respective instants, whether one has happened before another.

Another difference with our common perception of time is that several instants can be coincident, which means they ”happen” simultaneously. This is the case for instance when two successive events happen so close to each other that they cannot be distinguished by a given observer. In some concurrent languages, such as CCSL, this vision is completely embraced, since no instant can ”host” more than one event. This means that two events that seem to occur simultaneously will still be carried by different instants, but these instants will be coincident. This vision is closely linked to the notion of refinement, because it assumes that there exists no ultimate level of refinement on which an observer can know everything about the behavior of a system, since two coincident instants can always be distinguished when looking close enough to the execution of the system. Our relation of refinement heavily relies on this observation. Let us name this set of instants I.

2.2 Strict partial orders

As explained in the previous subsection, time cannot be seen as a single line that hosts any event occurrence. Instead, it contains a possibly infinite set of timelines that link instants that are observationally related. This means that the set of instants is not coupled with a total order but rather with a partial order that represents the knowledge the observer has of the behavior of the system. This means that each pair of instant is either:

- *strictly comparable*, through a precedence relation $\prec$
- *equivalent*, through a coincidence relation $\approx$
- *independent*, which means neither equivalent nor precedent

This is important to note that a partial order is not a single relation. It consists in two relations (the one defined above) that must fulfil certain properties, which are, as a reminder:

- $\approx$ is an equivalence relation
  - $\approx$ is reflexive: $\forall i \in I : i \approx i$
  - $\approx$ is transitive: $\forall (i,j,k) \in I^3 : i \approx j \wedge j \approx k \Rightarrow j \approx k$
  - $\approx$ is symmetrical: $\forall (i,j) \in I^2 : i \approx j \Rightarrow j \approx i$
- $\prec$ is irreflexive towards $\approx$: $\forall (i,j) \in I^2 : i \prec j \Rightarrow \neg(i \approx j)$
- $\prec$ is transitive: $\forall (i,j,k) \in I^3 : i \prec j \wedge j \prec k \Rightarrow j \prec k$
- $\prec$ respects $\approx$: $(\forall (i,j,k) \in I^3 : i \approx j \wedge i \prec k \Rightarrow j \prec k) \wedge (\forall (i,j,k) \in I^3 : i \approx j \wedge k \prec i \Rightarrow k \prec j)$

An example of strict partial order Let us consider the usual morning routine of Alice. She gets up then either takes her shower first then eats or the other way around. She always sings when she showers. After that, she takes off for work. The two possible traces depicting her behavior over a single day are depicted in Figure 1a and 1b. They consider the following set of possible events: getting up, showering, singing, eating and taking off, with their respecting aliases "up", "sho", "sin", "eat" and "off".

(a) A first possible behavior (b) A second possible behavior

Figure 1: Both possible behaviors

These possible behaviors are described by a time structure (derived from event structure [23, 40]) with an underlying partial order, that is depicted on Figure 2. The events "sho" and "eat" are concurrent and are not linked by any of the two relations composing the strict partial order. The blue vertical dashed line represents coincidence (when events occur simultaneously) while the red arrows represent precedence.

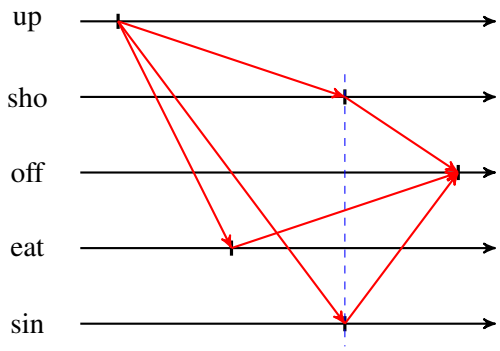


Figure 2: The underlying partial order

2.3 Our relation over strict partial orders

These reminders about strict partial orders and instants lead to the definition of the proposed refinement relation. As our approach is part of a denotational context, we need to express a relation between certain

data that are relevant to express refinement. These data cannot be the mere instants as these are not specific to a given execution, and these do not carry enough information. However, the strict partial orders binding them embed the necessary knowledge about the system behavior to be ordered in a way that respects the proposed time related instant refinement. Thus, we propose to instantiate these so-called data with the orders binding the instants together at a given level of observation. This binding of orders between instants and not instants themselves is the core contribution of this paper. The following relation takes two strict partial orders and states what it means for them to be in a relation of refinement.

Let Ω be the set of all sets : $\forall I \in \Omega, \forall (<_c, <_a, \approx_c, \approx_a) \in (I \times I)^4$:
$(<_c, \approx_c) <_r (<_a, \approx_a) \overset{d}{\iff} \forall (i_1, i_2) \in I$:

$$
\begin{aligned}
& i_1 <_c i_2 \Rightarrow i_1 <_a i_2 \vee i_1 \approx_a i_2 && (1)\\
\wedge\ & i_1 <_a i_2 \Rightarrow i_1 <_c i_2 && (2)\\
\wedge\ & i_1 \approx_c i_2 \Rightarrow i_1 \approx_a i_2 && (3)\\
\wedge\ & i_1 \approx_a i_2 \Rightarrow i_1 \approx_c i_2 \vee i_1 <_c i_2 \vee i_2 <_c i_1 && (4)
\end{aligned}
$$

In this definition, the level annotated by the index c is the lower (the more concrete) level of observation and a is the higher (the more abstract). We state what it means for a pair of relations to refine another pair of relations. We can only compare pairs of relations that are bounded to the same underlying set. This relation is composed of four predicates, each of which indicates how one of the four relations is translated into the other level of observation.

- *Precedence abstraction:* If a strictly precedes b in the lower level, then it can either be equivalent to it in the higher level or still precede it. This means that a distinction which is visible at a lower level can either disappear at a higher level or remain visible, depending on the behavior of the refinement around these instants.
- *Precedence embodiment:* If a strictly precedes b in the higher level, then it can only still precede it in the lower level. This means that the distinction between these instants was already existing in the higher level, and cannot be lost when refining. Looking closer to a system preserves precedence between instants.
- *Coincidence abstraction:* If a is equivalent to b in the lower level, it can only stay equivalent in the higher level. This means that looking at the system from a higher point of view cannot reveal temporal distinction between events.
- *Coincidence embodiment:* If a is equivalent to b in the higher level then the only thing we ensure is that these two instants are still related in the lower level. This means that both instants will still be related – they cannot become independent – but there is no guarantee on the nature of this relation.

This definition is coherent with CCSL point of view where instants can only hold one event. Two instants appearing coincident in a given level of refinement can potentially always be refined up to a point where a distinction appears, which justifies the fact that they should not be attached to the same physical instant.

3 A refinement example

The mathematical relation defined above aims at providing a formal construct to verify refinement between traces of execution. To illustrate its relevance, we propose to apply it to a simple example chosen for its simplicity and accuracy with respect to the idea of refinement. This is a simple system whose

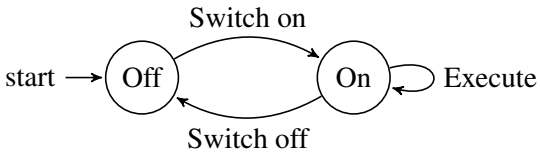


Figure 3: A simple system

behavior is represented as a transition system depicted on Figure 3. This system can be switched on and off. While it is on, an action can be executed any number of times. A possible trace – amongst an infinite number of them – of this system is depicted in Figure 4. t_{on}, t_{off} and t_{ex} respectively represent the occurrence of the "switch on", "switch off" and "execute" transitions.

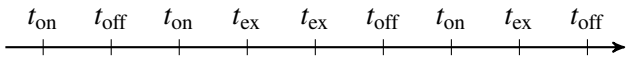


Figure 4: A trace on a single timeline

This trace starts with the birth of the system and possibly goes on indefinitely, which makes this representation partial. In addition, this design places each event on the same timeline, thus ignoring horizontal separation. In order to make it visible, we will represent, from now on, every different event on a specific timeline, such as on Figure 5. This approach is used in CCSL, where each timeline is represented by a clock which tracks the occurrences of a specific event. The instants on each timeline are totally ordered and those in the same vertical dashed blue lines are coincident.

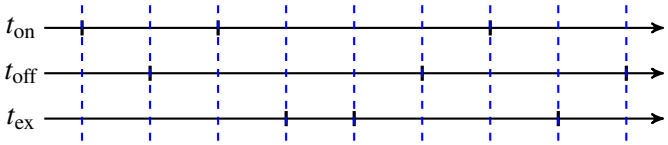


Figure 5: One timeline per event

The action executed by the system while running can be specified in various ways. We imagine here that our system is connected to a light through the use of a memory containing a variable x. This variable is assigned by our system to the values 1 or 0, and the light is turned on and off accordingly. When the system is switched on, the light remains down until a button is pressed which turns it on. Pressing the same button will alternatively turn it off and on. Shutting down the system turns it off. This behavior is depicted on Figure 6.

By specifying our system behavior, we defined events that can be added to its traces. t_{x_0} and t_{x_1} respectively correspond to the variable x being assigned 0 and 1. These additions belong to horizontal separation since we added a new part to our system (the module linked to the light). One of these possible traces is depicted in Figure 7. Some events are occurring simultaneously, for instance t_{on} always occurs on an instant coincident to an occurrence of t_{x_0}. Such relations between events can be defined in CCSL (a simple case of sub-clocking).

It is important to notice that when specifying the action executed by this system, we implicitly took a certain point of view. We deliberately ignored some lower level concerns such as the way a computer system handles a memory. This is where vertical separation takes place. Seeing closer to the machine will lead to other events which can refine the access to the variable x. For instance, the "switch on"

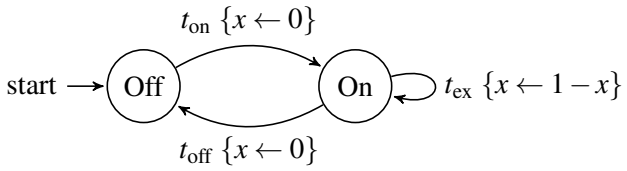


Figure 6: The system pilots a light

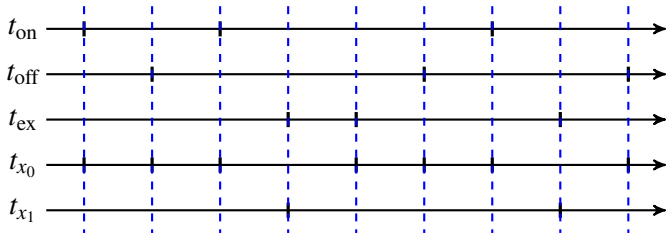


Figure 7: The trace of the system with the addition of the variable x

event can be viewed as a succession of actions, such as powering up the system, retrieving the address of x, computing (here there is no actual computation since 1 is an atomic value, but there could be in the case of a more complicated expression) the value of 1 and storing this value at the right address. These events, except for the first one, are used to handle the computation and the storing of a value in a memory. Taking into account these events require to view the system at a lower level than before, in which case its representation as a transition system is depicted in Figure 8.

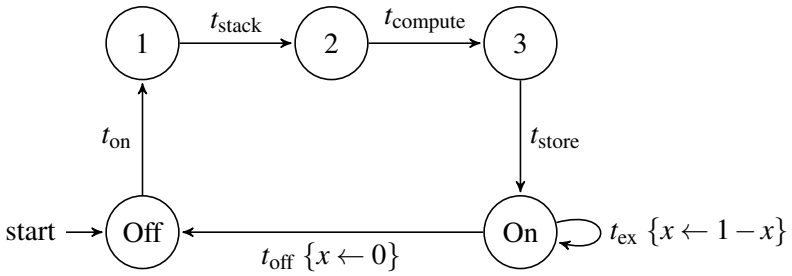
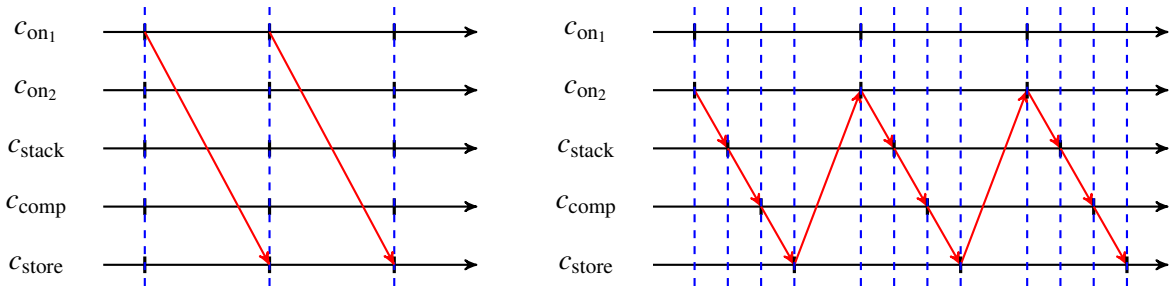


Figure 8: The refined system

The "switch on" transition has been refined in several transitions. t_{on} represents the powering of the system, t_{stack} the stacking of the address of x, $t_{compute}$ the computing of the value of the expression 1 and t_{store} the storing of the computed value at the stacked address. Note that we only refined one transition here for the sake of clarity and simplicity. Refining the other transitions would rely on exactly the same reasoning which is of no use for the relevance of this example.

This analyse induces two different points of view on our system. The higher level of observation is represented on Figure 9a. The events that are not refined are omitted from now on, for the sake of clarity. They don't influence the reasoning we are conducting, thus their omission is acceptable.

From the higher point of view, all the instants on which the sub-events occur are equivalent both to each other and to the containing event. Their underlying order is hidden and has no impact on the trace of the system at this level. The lower point of view, however, is different, as depicted on Figure 9b.

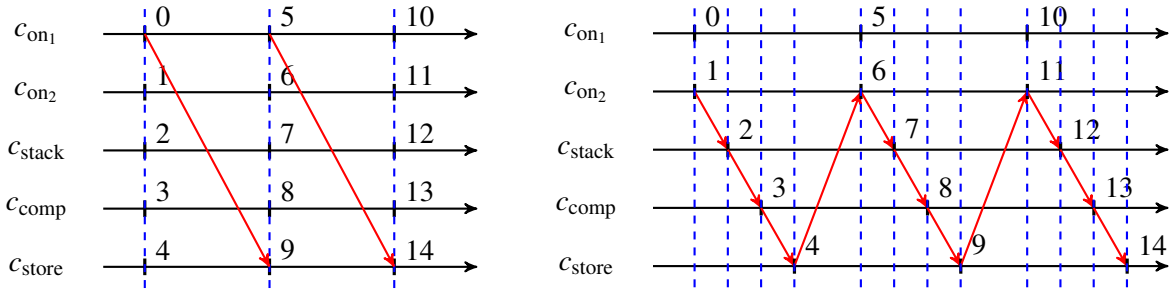


(a) The higher level of observation (b) The lower level of observation

Figure 9: Both levels of observation

For the lower level of observation, the different instants are ordered in a way such that they respect the specification in Figure 8. The blue dashed lines represents the equivalence classes induced by the respective partial orders while the red arrows represent the precedent relations of these orders (we did not represent the links that can be deduced by transitivity or other properties of partial orders).

Until now, the instants on which the events occur formed an unspecified set. Since our goal is to mechanize this example, we need to instantiate it to an actual set. We chose the natural numbers because they allow to annotate the traces while expressing quite easily the relations at both levels of refinement. The annotated higher level of observation is given in Figure 10a.



(a) The higher level annotated (b) The lower level annotated

Figure 10: Both annotated levels of observation

This representation allows us to define the coincidence and the precedence relations that bind its different instants, as subsets of $\mathbb{N} \times \mathbb{N}$. Since both these relations must be transitive, the coincidence must be symmetrical and they must form a strict partial order. We omit the related elements which can be deduced from these properties.

Coincidence Relation			**Precedence Relation**
(0 , 1)	(0 , 2)	(0 , 3)	(0 , 5)
(0 , 4)	(5 , 6)	(5 , 7)	
(5 , 8)	(5 , 9)	(10 , 11)	(5 , 10)
(10 , 12)	(10 , 13)	(10 , 14)	

Since the traces are infinite, there are an infinite number of couples in each relations. We only expressed them for the visible subset. We now define these relations for any natural number, by relying on euclidean decomposition of their operands by 5:

$$\forall (a,a') \in \mathbb{N}^2, \exists! \ (q,r,q',r') \in \mathbb{N}^4 \ : a = 5q + r \wedge r < 5 \wedge a' = 5q' + r' \wedge r' < 5$$

These relations, using the same notation, are defined as follow:

$$\forall (a,a') \in \mathbb{N}^2, a \approx_2 a' \Leftrightarrow q = q'$$
$$\forall (a,a') \in \mathbb{N}^2, a <_2 a' \Leftrightarrow q < q'$$

The same work can be achieved for the lower level of observation, which is displayed on Figure 10b. The relations extracted from Figure 10b are depicted in the table below. As previously explained, only the relevant couples are mentioned.

Coincidence Relation	**Precedence Relation**		
(0 , 1)	(1 , 2)	(2 , 3)	(3 , 4)
(5 , 6)	(4 , 5)	(6 , 7)	(7 , 8)
(10 , 11)	(8 , 9)	(9 , 10)	(11 , 12)
...	(12 , 13)	(13 , 14)	...

By taking the same decomposition as before, we can mathematically define the relations at the lower level of observation.

$$\forall (a,a') \in \mathbb{N}^2, a \approx_1 a' \Leftrightarrow (q_1 = q_2) \wedge ((r_1,r_2) \in [0,1]^2 \vee (r_1 = r_2 \wedge r_1 \notin [0,1]))$$
$$\forall (a,a') \in \mathbb{N}^2, a <_1 a' \Leftrightarrow (q_1 < q_2) \vee ((q_1 = q_2) \wedge (r_1 < r_2) \wedge (r_2 \neq 1))$$

Since both couples of relations have been defined mathematically, we can prove that they correspond to a situation of refinement. The proof has been done both on paper and in Agda, and is not presented here. It is however available on the first author's web page [1]. The steps in this proof are the following:

- Prove that the two couples of relation form strict partial orders (12 predicates).
- Prove that these orders satisfy the refinement relation (4 predicates).

4 Mechanization of the refinement relation

This work is supported by a significant effort of mechanization. We advocate that any proof and formalization should be done through formal methods in order both to ease and verify the mathematical content of the work. In our case, this effort has been done using a proof assistant called Agda. We briefly present it in this section before getting to the benefits of this mechanization.

4.1 Agda

Agda is a dependently typed programming language developed by Ulf Norell at Chalmers University. As any other language, the types of which can depend on values, it is expressive enough to build mathematical theories, thanks to the Curry-Howard isomorphism, which ensures the correctness of any property whose equivalent type is inhabited. The core of the language is an intuitionist type theory, on which the well-known tool Coq is based as well. Although these two languages share the same heart, they are quite different when it comes to developing and proving properties. Coq uses named tactics, the action of which is hidden from the reader of the Coq file – as well as the underlying lambda-terms – while

[1] `http://montin.perso.enseeiht.fr`

Agda provides a framework to help the programmer write them by hand, thus making them visible in the Agda file. This framework is what makes programming in Agda possible since typed lambda terms are arguably impossible to write without software assistance, assuming their type reaches a certain level of complexity.

Agda also differs from Coq by its native unification mechanism, which is usually summarized by "Agda allows to pattern-match on equality proofs". Although unification can hardly be reduced to this simple sentence, Agda indeed allows to case-split on the equality proofs, thus unifying the operands of the equality. More generally, Agda is able to infer, by unification, the value of variables present in the context of a proof. Coq does not provide such a straight-forward mechanism and handles cases usually solved by unification in Agda with other ways that we find less convenient.

The rest of this paper contains small pieces of Agda code, depicting either data structures, predicates or proofs established during our development. Although these blocks help assessing the technical aspects of our work, their understanding is not mandatory to grasp the notions we describe and the reasoning behind them. Their goal is to briefly picture what Agda proofs look like and to help the reader assess the underlying effort of this work. Here is our relation written in Agda:

```
_≺≈_ : ∀ {ℓ} → Rel (Rel A ℓ × Rel A ℓ) _
(_≈₁_ , _≺₁_) ≺≈ (_≈₂_ , _≺₂_) =
  (∀ {a b} → a ≺₁ b → a ≺₂ b ⊎ a ≈₂ b) ×
  (∀ {a b} → a ≺₂ b → a ≺₁ b) ×
  (∀ {a b} → a ≈₂ b → a ≈₁ b ⊎ a ≺₁ b ⊎ b ≺₁ a) ×
  (∀ {a b} → a ≈₁ b → a ≈₂ b)
```

4.2 Properties of the refinement relation

It looks reasonable to assume that our refinement relation should be a strict partial order between strict partial orders. Being able to prove such property would enforce the correctness of our definition towards the refinement requirement. However, as we mentioned earlier, a strict partial order is based on an equivalence relation. This relation could be the propositional equality, or another relation that we defined. We tried both possibilities, and we present the results of these attempts in this section.

4.2.1 It is a pre-order towards propositional equality

As a reminder, a pre-order is an algebraic structure composed of an equivalence relation and a precedence relation which is transitive and reflexive according to the equivalence relation. We showed that our refinement relation formed a pre-order towards the propositional equality. The propositional equality, in dependent types, is a family of types generated by the reflexivity rule. This means that two quantities are propositionally equal if they were built with the same constructors. We start by proving that our relation is transitive:

```
trans≺≈ : ∀ {ℓ} → Transitive (_≺≈_ {ℓ})
trans≺≈ p q {a} {b} with p {a} {b} | q {a} {b} | p {b} {a}
trans≺≈ p q {a} {b} | pr₁ , pr₂ , pr₃ , pr₄ | pr₅ , pr₆ , pr₇ , pr₈ | _ , pr₁₀ , _ , _
  = (λ x → case pr₁ x of (λ {(inj₁ x₁) → pr₅ x₁ ; (inj₂ y) → inj₂ (pr₇ y)})) ,
    (λ x → pr₂ (pr₆ x)) , (λ x → pr₇ (pr₃ x)) , (λ x → case pr₈ x of (λ {(inj₁ x₁) → pr₄ x₁
    ; (inj₂ (inj₁ x₁)) → inj₂ (inj₁ (pr₂ x₁)) ; (inj₂(inj₂ y)) → inj₂ (inj₂ (pr₁₀ y))}))
```

We also prove it is reflexive:

```
refl≺≈ : ∀ {ℓ} → Reflexive (_≺≈_ {ℓ})
refl≺≈ = (λ x → inj₁ x) , (λ x → x) , (λ z → z) , (λ x → inj₁ x)
```

This allows us to exhibit the pre-order we aimed for.

```
preorder≺≈≡ : ∀ {ℓ} → IsPreorder _≡_ (_≺≈_ {ℓ})
preorder≺≈≡ = record { isEquivalence = isEquivalence
  ; reflexive = λ {i} {j} x → case x of (λ {refl → refl≺≈ {x = i}}) ; trans = trans≺≈ }
```

4.2.2 It is a partial order towards the equivalence between relations

Two relations are equivalent when the subset they form are equal. We implemented this definition for our couples of relations:

```
_≈≈_ : ∀ {ℓ} → Rel (Rel A ℓ × Rel A ℓ) _
(_≈₁_ , _≺₁_) ≈≈ (_≈₂_ , _≺₂_) = ∀ {a b} →
  (a ≈₁ b → a ≈₂ b) ×
  (a ≈₂ b → a ≈₁ b) ×
  (a ≺₁ b → a ≺₂ b) ×
  (a ≺₂ b → a ≺₁ b)
```

A partial order is a pre-order with an anti-symmetrical property between its two underlying relations. We already proved that our refinement relation was transitive, but we still need to prove that our equivalence relation is indeed an equivalence and that the properties of reflexivity and antisymmetry hold between them. The equivalence is obvious and is not presented here. The reflexivity is proved as follow:

```
refl≺≈≈ : ∀ {ℓ} → (_≈≈_ {ℓ}) ⇒ (_≺≈_ {ℓ})
refl≺≈≈ x {a} {b} with x {a} {b}
refl≺≈≈ x {a} {b} | proj₃ , proj₄ , proj₅ , proj₆ =
  (λ x₁ → inj₁ (proj₅ x₁)) , (λ x₁ → proj₆ x₁) , (λ x₁ → proj₃ x₁) , (λ x₁ → inj₁ (proj₄ x₁))
```

As for the antisymmetry:

```
antisym≺≈ : ∀ {ℓ} → Antisymmetric _≈≈_ (_≺≈_ {ℓ})
antisym≺≈ x x₁ {a} {b} with x {a} {b} | x₁ {a} {b}
antisym≺≈ x₁ x₂ {a} {b} | proj₃ , proj₄ , proj₅ , proj₆ | proj₇ , proj₈ , proj₉ , proj₁₀
  = (λ x → proj₅ x) , (λ x → proj₉ x) , (λ x → proj₈ x) , (λ x → proj₄ x)
```

This allows us to exhibit the partial order we aimed for:

```
partialOrder≺≈≈ : ∀ {ℓ} → IsPartialOrder _≈≈_ (_≺≈_ {ℓ})
partialOrder≺≈≈ = record
  { isPreorder = record { isEquivalence = equiv≈≈ ; reflexive = refl≺≈≈ ; trans = trans≺≈ }
  ; antisym = antisym≺≈ }
```

4.3 It preserves CCSL operators

CCSL denotational semantics: In a previous work, we mechanized the denotational semantics of CCSL in Agda. This section gives the required notions about this mechanization in order to connect it to our refinement relation.

CCSL is based on clocks, which represents the different occurrences of a specific event. Typically, a clock represents one of the different timelines we depicted in the different figures in this paper. In our work, we represent clocks by a record containing a predicate to emulate the subset of instants on which this clock ticks, and a predicate which makes sure the ticks of the clocks are totally ordered regarding the given strict partial order:

```
record Clock : Set₁ where
  constructor
    clock
  field
    Ticks  : Pred Support lzero
    TicTot : _≺_ isTotalFor Ticks
```

CCSL provides several constructs to constrain the different clocks of a system amongst each other. They are grouped into two different categories: the relations and the expression. A relation is a direct constraint between two clocks, while an expression, in our denotational semantics, is a predicate over three clocks:

```
Relation : Set₁
Relation = Clock → Clock → Set
```

```
Expression : Set₁
Expression = Clock → Clock → Clock → Set
```

The goal of this paper is not to detail the whole semantics, hence we only give one example of each of these categories. The relation we present is the sub-clocking. A clock c_1 is a sub-clock of a clock c_2 when $T(c_1) \subset T(c_2)$:

```
_⊑_ : Relation
(clock Tc₁ _) ⊑ (clock Tc₂ _) = ∀ {x₁} → x₁ ∈ Tc₁ → ∃ \x₂ → x₁ ≈ x₂ × x₂ ∈ Tc₂
```

The expression we will present is the union. A clock c is considered the union of a clock c_1 and a clock c_2 when $T(c) = T(c_1) \cup T(c_2)$:

```
_≡_∪_ : Expression
clock Tc _ ≡ clock Tc₁ _ ∪ clock Tc₂ _ =
    (∀ {i} → (Tc₁ i ⊎ Tc₂ i) → ∃ \j → i ≈ j × Tc j)
    × (∀ {i} → Tc i → ∃ \j → i ≈ j × (Tc₁ j ⊎ Tc₂ j))
```

A relation between clocks: This clock definition allows extending our refinement relation to clocks. Informally, a clock refines another one when it represents a thinner event which was hidden by the first clock. For instance, if we get back to our example, the "switch on" clock is refined by several clocks, including the "compute" one. Let us consider the following definition:

```
_refc_ : REL (Clock _) (Clock _) _
(clock Ticks₁ _) refc (clock Ticks₂ _) =
  × (∀ {x} → Ticks₂ x → ∃ λ y → Ticks₁ y × (y ≈₂ x))
  × (∀ {x} → Ticks₁ x → ∃ λ y → Ticks₂ y × (y ≈₂ x))
```

A clocks refines another if they are defined on refined partial orders, while also obeying the following predicates: each tick of the more abstract clock is refined by at least one tick of the concrete clock and each tick of the concrete clock is the refinement of a tick of the abstract clock.

Proofs of semantic preservation: We prove the preservation of the semantics of the CCSL constructs towards the refinement relation. This preservation is described and discussed about the two semantic elements we presented, the sub-clocking and the union. The proofs are not presented here because they are not relevant but they are available online. The preservation property about sub-clocking is as follows: given four clocks c_a, c_b, c_1, c_2, if c_1 is a sub-clock of c_2, if c_1 refines c_a and c_2 refines c_b then c_a is a sub-clock of c_b.

```
subclockingRefinement : c₁ ⊑₁ c₂ → c₁ ≺refc c₁₁ → c₂ refc c₂₂ → c₁₁ ⊑₂ c₂₂
```

The preservation property about union is a follows: given four clocks c_0 c_1, c_2 and c, if c_1 refines c, if c_2 refines c and if $c_0 = c_1 \cup c_2$ then c_0 refines c.

```
unionRefinement : c₁ refc c → c₂ refc c → c₀ ≡ c₁ ∪ c₂ → c₀ refc c
```

5 Conclusion

5.1 Assessment

This contribution provided a refinement relation for time models that allows system developers focusing on their own view of the system rather than a common view shared among each of them. This enables seizing their constraints even better without taking into account considerations from other levels of observation. The constraints on the system can then be described and solved at all different levels with the assurance that none of them will be compromising the others. Furthermore, instant refinement can be used to specify simulations and bisimulations (or mostly weak bisimulations) between systems. In this case, the two specifications would not be different observation levels of the same system, but different ways of specifying its behavior, or different systems that must satisfy the same interface.

More precisely, this paper presented a relation over strict partial orders whose goal is to model instant refinement. Each level of abstraction is represented by a specific strict partial order while keeping the link between them. This definition is mechanized in Agda, which allowed us to prove different algebraic properties about it as well as connecting it to the mechanization of CCSL made in a previous work. The bridge between these two works allowed us to prove the preservation of several CCSL operators through our relation of refinement.

5.2 Future works

Several future works are currently being conducted:

- Ultimately, we would like to allow the system developers to express their constraints at the most suitable level of abstraction. This could only be done if their constraints are propagated in the other levels where other constraints are specified. Thus, we plan to complete the link between CCSL and our instant refinement through the proof of multiple preservation properties. This extension could also lead to automated reduction of constraints sets relying on additional properties about CCSL operators.
- The CCSL team at INRIA[2] plans to integration our notion of refinement to their toolsets. While refinement cannot be considered as yet another CCSL operator, it could still be used to provide more expressiveness through the use of several partial orders instead of a single one.
- We would like to apply our approach to a more complex example. In that purpose, we would like to refactor and complete our previous work regarding the proof of correctness of a translation between langages (process models to Petri nets) as a weak bisimulation that binds these languages with our refinement relation. Indeed, weak bisimulation can be seen as a special case of refinement which we would like to investigate.

[2]We thank the CCSL team at INRIA for the fruitful discussions we had around CCSL and the need for a refinement relation.

References

[1] Martín Abadi & Leslie Lamport (1991): *The Existence of Refinement Mappings*. Theor. Comput. Sci. 82(2).

[2] Jean-Raymond Abrial (2005): *The B-book - assigning programs to meanings*. Cambridge University Press.

[3] Jean-Raymond Abrial (2010): *Modeling in Event-B - System and Software Engineering*. Cambridge University Press.

[4] Jean-Raymond Abrial, Dominique Cansell & Dominique Méry (2005): *Refinement and Reachability in Event$_B$*. In Helen Treharne, Steve King, Martin C. Henson & Steve A. Schneider, editors: *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, Lecture Notes in Computer Science 3455, Springer, pp. 222–241.

[5] Charles André & Frédéric Mallet (2008): *Clock Constraints in UML/MARTE CCSL*. Research Report RR-6540, INRIA.

[6] Ralph-Johan Back (1989): *Refinement Calculus, Part II: Parallel and Reactive Programs*. In: *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop, Mook, The Netherlands, May 29 - June 2, 1989, Proceedings*, pp. 67–93.

[7] Ralph-Johan Back & Joakim von Wright (1989): *Refinement Calculus, Part I: Sequential Nondeterministic Programs*. In: *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop, Mook, The Netherlands, May 29 - June 2, 1989, Proceedings*, pp. 42–66.

[8] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series.

[9] Ana Bove & Peter Dybjer (2008): *Dependent Types at Work*. In: *Language Engineering and Rigorous Software Development, Intl. LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, pp. 57–99.

[10] Manfred Broy (2001): *Refinement of time*. Theor. Comput. Sci. 253(1), pp. 3–26.

[11] Joseph T. Buck, Soonhoi Ha, Edward A. Lee & David G. Messerschmitt (1994): *Ptolemy: A Framework for Simulating and Prototyping Heterogenous Systems*. Int. Journal in Computer Simulation 4(2).

[12] Benoît Combemale, Julien DeAntoni, Benoit Baudry, Robert B. France, Jean-Marc Jézéquel & Jeff Gray (2014): *Globalizing Modeling Languages*. IEEE Computer 47(6).

[13] Benoît Combemale, Julien DeAntoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry & Robert B. France (2013): *Reifying Concurrency for Executable Metamodeling*. In: *Software Language Engineering - 6th Intl. Conf., SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proc.*

[14] Julien Deantoni, Charles André & Régis Gascon (2014): *CCSL denotational semantics*. Research Report RR-8628.

[15] Julien DeAntoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau & Benoît Combemale (2015): *Towards a meta-language for the concurrency concern in DSLs*. In: *Proc. of the 2015 Design, Automation & Test in Europe Conf. & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015.*

[16] Julien Deantoni & Frédéric Mallet (2012): *TimeSquare: Treat your Models with Logical Time*. In: *TOOLS - 50th Intl. Conf. on Objects, Models, Components, Patterns - 2012.*

[17] Manuel Garnacho, Jean-Paul Bodeveix & Mamoun Filali-Amine (2013): *A Mechanized Semantic Framework for Real-Time Systems*. In: *Formal Modeling and Analysis of Timed Systems - 11th Intl. Conf., FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proc.*

[18] Mike Gemünde, Jens Brandt & Klaus Schneider (2013): *Clock refinement in imperative synchronous languages*. EURASIP J. Emb. Sys. 2013.

[19] Roger Hale, Rachel Cardell-Oliver & John Herbert (1993): *An Embedding of Timed Transition Systems in HOL*. Formal Methods in System Design 3(1/2).

[20] Cécile Hardebolle & Frédéric Boulanger (2007): *ModHel'X: A Component-Oriented Approach to Multi-Formalism Modeling*. In: *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*.

[21] Jifeng He (1989): *Process Simulation and Refinement*. *Formal Asp. Comput.* 1(3), pp. 229–241.

[22] Wim H. Hesselink (2011): *Simulation refinement for concurrency verification*. *Sci. Comput. Program.* 76(9), pp. 739–755.

[23] Leslie Lamport (1978): *Time, Clocks, and the Ordering of Events in a Distributed System*. *Commun. ACM* 21(7), pp. 558–565.

[24] Matias Ezequiel Vara Larsen, Julien DeAntoni, Benoît Combemale & Frédéric Mallet (2015): *A Behavioral Coordination Operator Language (BCOoL)*. In: *18th ACM/IEEE Intl. Conf. on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*.

[25] Florent Latombe, Xavier Crégut, Benoît Combemale, Julien DeAntoni & Marc Pantel (2015): *Weaving concurrency in executable domain-specific modeling languages*. In: *Proc. of the 2015 ACM SIGPLAN Intl. Conf. on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*.

[26] Nancy A. Lynch & Frits W. Vaandrager (1995): *Forward and Backward Simulations: I. Untimed Systems*. *Inf. Comput.* 121(2), pp. 214–233.

[27] Nancy A. Lynch & Frits W. Vaandrager (1996): *Forward and Backward Simulations, II: Timing-Based Systems*. *Inf. Comput.* 128(1), pp. 1–25.

[28] Jan Malakhovski: *Brutal [Meta]Introduction to Dependent Types in Agda*.

[29] Louis Mandel, Cédric Pasteur & Marc Pouzet (2015): *Time refinement in a functional synchronous language*. *Sci. Comput. Program.* 111.

[30] Per Martin-Löf (1972): *An Intuitionistic Theory of Types*. Available at `http://archive-pml.github.io/martin-lof/pdfs/An-Intuitionistic-Theory-of-Types-1972.pdf`.

[31] Per Martin-Löf (1984): *Intuitionistic type theory. Notes by Giovanni Sambin. Studies in Proof Theory*. Available at `http://archive-pml.github.io/martin-lof/pdfs/Bibliopolis-Book-1984.pdf`.

[32] Jan Mikac & Paul Caspi (2005): *Temporal refinement for Lustre*. In: *Proc. of the 5th Intl. Workshop on Synchronous Languages, Applications and Programs, Edimburg, April 2005*.

[33] Ulf Norell (2009): *Dependently typed programming in Agda*. In: *Proc. of TLDI'09: 2009 ACM SIGPLAN Intl. Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*.

[34] Christine Paulin-Mohring (2001): *Modelisation of Timed Automata in Coq*. In: *Theoretical Aspects of Computer Software, 4th Intl. Symp., TACS 2001, Sendai, Japan, October 29-31, 2001, Proc.*

[35] Amir Pnueli (1977): *The Temporal Logic of Programs*. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, IEEE Computer Society, pp. 46–57.

[36] Steve Reeves & David Streader (2003): *Comparison of Data and Process Refinement*. In Jin Song Dong & Jim Woodcock, editors: *Formal Methods and Software Engineering, 5th International Conference on Formal Engineering Methods, ICFEM 2003, Singapore, November 5-7, 2003, Proceedings*, *Lecture Notes in Computer Science* 2885, Springer, pp. 266–285.

[37] Steve Reeves & David Streader (2008): *General Refinement, Part One: Interfaces, Determinism and Special Refinement*. *Electr. Notes Theor. Comput. Sci.* 214, pp. 277–307.

[38] Willem P. de Roever & Kai Engelhardt (1998): *Data Refinement: Model-oriented Proof Theories and their Comparison*. *Cambridge Tracts in Theoretical Computer Science* 46, Cambridge University Press.

[39] Jean-Pierre Talpin, Paul Le Guernic, Sandeep K. Shukla, Frederic Doucet & Rajesh K. Gupta (2004): *Formal Refinement Checking in a System-level Design Methodology*. *Fundam. Inform.* 62(2), pp. 243–273.

[40] Glynn Winskel (1986): *Event Structures*. In: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proc. of an Advanced Course, Bad Honnef, 8.-19. September 1986*.