Ordering strict partial orders to model behavioural refinement

Mathieu Montin[1,2] and Marc Pantel[1,2]

[1] Université de Toulouse ; Toulouse INP, *IRIT*
2 rue Camichel, BP 7122, 31071 Toulouse Cedex 7, France
[2] *CNRS* ; Institut de Recherche en Informatique de Toulouse (IRIT)
Toulouse, France

Abstract. Software is now ubiquitous and involved in complex interactions with the human users and the physical world in so-called cyber-physical systems (Cps). Separation of concerns is thus a key issue 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. This contribution aims at providing a formal construct for the verification of vertical separation in time models, through the definition of an order between strict partial orders used to relate the different instants in asynchronous systems. 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.

Keywords: Instant refinement, partial orders, Agda

1 Introduction

1.1 Separation of concerns

Software is now ubiquitous and involved in complex interactions with the human users and the physical world in so-called Cyber-Physical Systems (Cps). Since these systems are increasingly dense and complex, separation of concerns is a key issue in their development. There exists many kinds of separation of concerns in Cps development : 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 [4, 25]. A second one corresponds to the various components in the system architecture at a given level of refinement and is called horizontal.

The horizontal separation at design time is usually handled through the expression of the various system parts in different Domain Specific Modelling Languages (Dsml), the execution of which, for validation and 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 behaviour of the systems. This heterogenous modelling approach has been integrated in the Ptolemy toolset proposed by Lee et al. [7], the ModHel'X toolset proposed by Boulanger et al. [16] and the Gemoc studio proposed by Combemale et al. [8].

As for the vertical separation, it 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 [1, 2] in order to prove the preservation of the properties from the specification to the implementation. In the case of asynchronous systems, refinement corresponds to replacing τ transition by effective actions. In the case of synchronous systems, refinement correspond to decomposing an instant at a given level to 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 [24] and then as time refinement for reactive languages [14, 21].

Our work takes place in Gemoc that mixes both horizontal and vertical separation of concerns. Indeed, Gemoc allows to define the various Dsml used to model the various parts in a Cps in the various phases of the development. Thus, Dsml are combined both in an horizontal and vertical manners. Gemoc relies on the UML MARTE CCSL (Clock Constraint Specific Language) in order to model both the MoC for the various Dsml [9, 11, 18, 19] and the coordination between Dsml using the Behavioural Coordination Language (BeCool) [17]. Our seminal work in Gemoc targeted the horizontal separation of concerns. This contribution targets the vertical separation of concerns. More precisely, we want to assess the relations between the various models of the same system part in a vertical separation of concerns. In that purpose, we need to provide a mechanized definition for the vertical relation and apply it to CCSL which is at the core of the concurrent part of Gemoc.

This work handles this issue through the introduction of 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 vertical separation in these models, through the definition of an order relation between these partial orders.

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 joined to a denotational semantics of CCSL, which has already been mechanized using the same language and tool. This allows to assess the properties of this new relation and prove that it preserves the semantics of the different CCSL operators. This contribution relies on a thorough refinement example corresponding to classical oversampling in synchronous systems.

1.2 Additional related works

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 could thus be integrated with any other concurrent languages. Formal mechanization of temporal languages has already been done using other formal methods, for example [15] uses Higher Order Logic in Isabelle/HOL; [13] and [27] use the Calculus of Inductive Constructions in Coq, see [5]. 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 [26], [20] and [6]. Although Agda differs from Coq by several aspects, both of these tools rely on the same underlying intuitionist type theory, first described in [22] and clarified in [23]. The paper version of CCSL denotational semantics, which is connected to this work, can be found in [10]. TimeSquare, the tool developed to describe CCSL systems as well as solve constraint sets has been presented in [12]. As for CCSL itself, it was first presented in [3].

1.3 Denotational semantics

Our work aims at being integrated in a denotational environment where CCSL heart has been mechanized. This integration has already begun through the proof of several properties of preservation regarding CCSL operators through our notion of refinement, that will be described later on in this paper. This means that our notion of refinement must be compatible with a denotational world. Usually, this separation of concerns is deployed in operational environments, where instants are explicitly added to an existing set when a new layer of refinement is added to the current description of the system. This vision is purely operational because it requires an actual computation to make this addition. In a denotational semantics however, such an accretion of events cannot be considered since we remain descriptive toward the notions we manipulate. Let's assume there exists a set $E1$ containing instants on which events occur. Adding a layer of refinement induces the addition of new events occurring over new instants. These instants can be regrouped in a set $E2$. The global set of instants is then $E = E1 \cup E2$. In operational semantics, instants are just created then added to the already existing set $E1$ while the new level of refinement is described and computed. In a denotational point of view, all instants have always existed, regardless of whether they belong to a given layer of refinement or another. The denotational semantics describes the link between the different levels, rather than computing them.

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 operational semantics as they naturally embed the operation of accretion through their 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 expressivity to mechanize denotational semantics. This justifies the use of Agda for this work where we remain descriptive and never actually compute the traces of events on which our relation is ultimately defined. Ultimately, our work should be coupled with operational semantics through proofs of preservation.

1.4 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 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 summarised 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.

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 understand the notions we describe and the reasoning behind them. They only represent a small part in this paper and their presence is justified by the underlying effort of mechanization present in this work.

2 Time and refinement

This section introduces notions inherent to time handling in asynchronous systems, from the instants to the relation of refinement between strict partial orders. The sections regarding instants and partial orders summarize common knowledge among researchers in the field of time handling in such systems, while the section on refinement defines the relation of refinement we propose.

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 depicted as a single timeline 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 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. 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 behaviour 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.

In our Agda framework, instants will be represented by a generic set, the properties of which are unspecified. It exists and relations can be defined on top of it. We call this set *Support*, and we leave the name *Instant* to the algebraic structure which will be described in the next subsection. The cardinality of this set also remains unspecified (we do not make assumptions on whether it is countable or not prior to actual examples). It Agda, `Set` represents a type, and not a mathematical set. In dependently typed languages, sets can be emulated as unary relations (predicates over a type) but are not native constructs of the languages. However we will often abusively call an Agda `Set` a set.

```
Support : Set
```

2.2 Strict partial orders

As explained in the previous subsection, time cannot be seen as a single line on which events occur. 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 behaviour 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

To form a partial order, these relations must fulfil certain properties, which are, as a reminder :

- $\approx$ is an equivalence relation
- $\prec$ is irreflexive regarding $\approx$
- $\prec$ is transitive
- $\prec$ respects the equivalence classes induced by $\approx$

The Agda library provides such an algebraic structure.

```
record IsStrictPartialOrder {a ℓ₁ ℓ₂} {A : Set a}
  (_≈_ : Rel A ℓ₁) (_<_ : Rel A ℓ₂) : Set (a ⊔ ℓ₁ ⊔ ℓ₂)
    where field
      isEquivalence : IsEquivalence _≈_
      irrefl        : Irreflexive _≈_ _<_
      trans         : Transitive _<_
      <-resp-≈      : _<_ Respects₂ _≈_
```

This record contains the four properties described earlier, which ensure the two relations indeed form a strict partial order. The predicates `IsEquivalence`, `Irreflexive`, `Transitive`, `Respects` are defined in the standard library as well. They are not detailed here, as their functions are straightforward to understand.

2.3 A relation over strict partial orders

These reminders about strict partial orders and instants lead us to the definition of the relation of refinement. Since our approach is part of a denotational context, we need to express a relation between certain "quantities" that are relevant to express refinement. These quantities cannot be the instants themselves because they are not specific to a given execution, and they do not carry enough information. However, the strict partial orders binding them embed the necessary knowledge about the system behaviour to be ordered in a way that respects the proposed time related instant refinement. Thus, we propose to instantiate these so-called "quantities" 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, expressed in Agda (in a way that is very close to the paper version) takes two pairs of orders and expresses what it means that one pair refines the other. These pairs of orders aim at being encapsulated in the previous definition of strict partial orders.

Let us consider the following Agda relation.

```
_≺≈_ : ∀ {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)
```

In this definition, the level annotated by the index 1 is the lower (the more concrete) level of observation and 2 is the higher. We state what it means for a pair of relations to refine another pair of relation. Note that these relations are defined on a set (here A) which will eventually be the instants. 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 indicate how one of the four relations will be translated into the other level of observation.

- 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 behaviour of the refinement around these instants.
- However, 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.
- 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.
- 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.

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 refinement up to a point where a distinction appears, which justifies the fact that they should not be attached to the same physical instant. Our definition can be extended to strict partial order. A strict partial order refines another when their underlying relations satisfy our relation of refinement. This enlargement is purely syntactic.

3 A refinement example

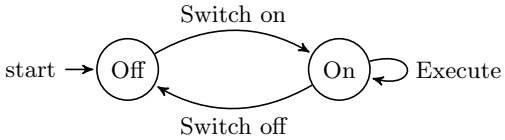


Fig. 1: A simple system

This section presents a system example upon which our relation of refinement can be applied. This is a simple system whose behaviour is represented as a

transition system depicted on Figure 1. 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 2. t_{on}, t_{off} and t_{ex} respectively represent the occurrence of the "switch on", "switch off" and "execute" transitions.

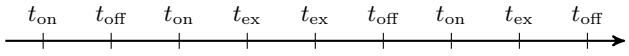


Fig. 2: A trace on a single timeline

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 3. 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.

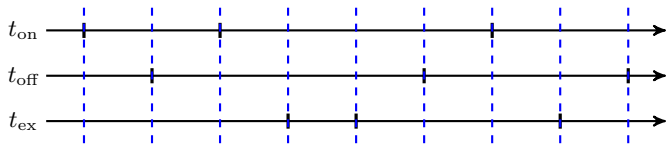


Fig. 3: One timeline per event

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 behaviour is depicted on Figure 4.

By specifying our system behaviour, 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 5. 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

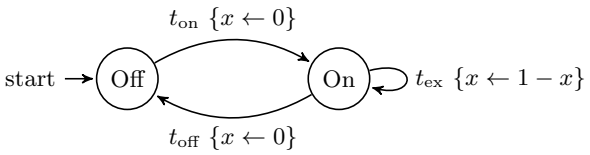


Fig. 4: The system pilots a light

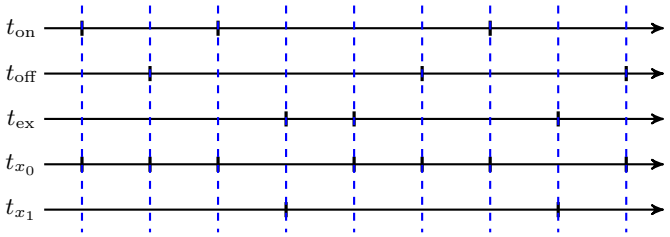


Fig. 5: The trace of the system with the addition of the variable x

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" 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 6.

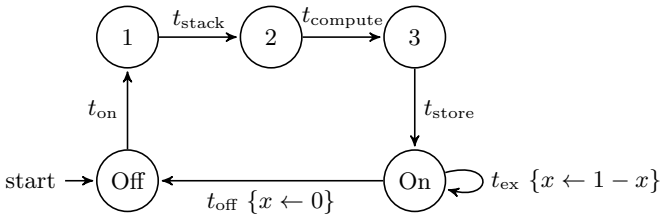


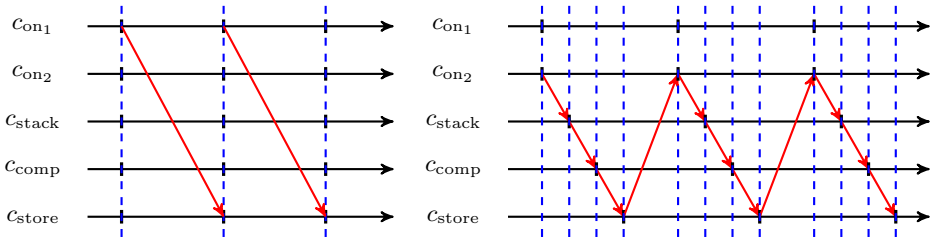
Fig. 6: The refined system

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 com-

puted 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 7a. 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.

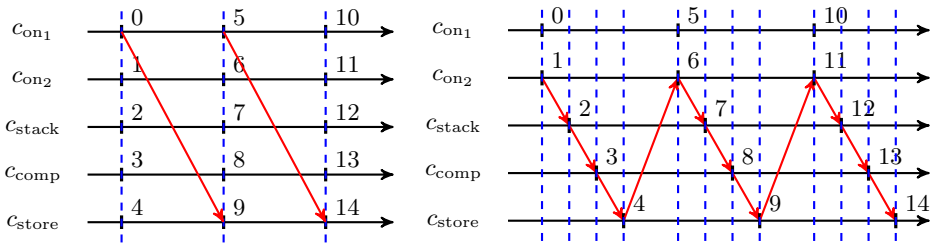
From the higher point of view, all the instants on which the sub-events occur are equivalent both to each other and to the containing event. Their underlying order is hidden and has no impact on the trace of the system at this level. The lower point of view, however, is different, as depicted on Figure 7b.



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

Fig. 7: 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 6. 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).



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

Fig. 8: Both annotated levels of observation

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 8a.

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 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 8b. The relations extracted from Figure 8b 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 available on the first author's web page [3]

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

4 Properties about the refinement relation

4.1 Mathematical properties

This section gives properties about the refinement relation we propose. The proof for the properties are omitted, but are available on the first author's web page with the whole Agda development about refinement and CCSL.

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.

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 proved that our relation of refinement form a partial order with this equivalence.

4.2 Refinement and CCSL

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 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

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 we made in a previous work. The bridge between these two works has allowed us to prove the preservation of several CCSL operators through our relation of refinement. This work will be followed by several future works including:

- The extension of our link between CCSL and instant refinement through the proof of additional preservation properties.
- The integration of instant refinement in CCSL and its associated toolset in cooperation with the CCSL team at INRIA.
- The refactoring of a previous work regarding the weak bi-simulation between Petri Net and models of processes (xSPEM) to prove language refinement between xSPEM and their encoding in Petri Nets.

References

1. Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.
2. Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
3. Charles André and Frédéric Mallet. Clock Constraints in UML/MARTE CCSL. Research Report RR-6540, INRIA, 2008.
4. Ralph-Johan Back. On correct refinement of programs. *J. Comput. Syst. Sci.*, 23(1), 1981.
5. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. 2004.
6. Ana Bove and Peter Dybjer. 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*, pages 57–99, 2008.
7. Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2), 1994.
8. Benoît Combemale, Julien DeAntoni, Benoit Baudry, Robert B. France, Jean-Marc Jézéquel, and Jeff Gray. Globalizing modeling languages. *IEEE Computer*, 47(6), 2014.
9. Benoît Combemale, Julien DeAntoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert B. France. Reifying concurrency for executable metamodeling. In *Software Language Engineering - 6th Intl. Conf., SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proc.*, 2013.
10. Julien Deantoni, Charles André, and Régis Gascon. CCSL denotational semantics. Research Report RR-8628, 2014.

11. Julien DeAntoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau, and Benoît Combemale. 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*, 2015.
12. Julien Deantoni and Frédéric Mallet. TimeSquare: Treat your Models with Logical Time. In *TOOLS - 50th Intl. Conf. on Objects, Models, Components, Patterns - 2012*, 2012.
13. Manuel Garnacho, Jean-Paul Bodeveix, and Mamoun Filali-Amine. 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.*, 2013.
14. Mike Gemünde, Jens Brandt, and Klaus Schneider. Clock refinement in imperative synchronous languages. *EURASIP J. Emb. Sys.*, 2013, 2013.
15. Roger Hale, Rachel Cardell-Oliver, and John Herbert. An embedding of timed transition systems in HOL. *Formal Methods in System Design*, 3(1/2), 1993.
16. Cécile Hardebolle and Frédéric Boulanger. 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*, 2007.
17. Matias Ezequiel Vara Larsen, Julien DeAntoni, Benoît Combemale, and Frédéric Mallet. 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*, 2015.
18. Florent Latombe, Xavier Crégut, Benoît Combemale, Julien DeAntoni, and Marc Pantel. 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*, 2015.
19. Florent Latombe, Xavier Crégut, Julien DeAntoni, Marc Pantel, and Benoît Combemale. Coping with semantic variation points in domain-specific modeling languages. In *Proc. of the 1st Intl. Workshop on Executable Modeling, (MODELS 2015), Ottawa, Canada, September 27, 2015.*, 2015.
20. Jan Malakhovski. Brutal [meta]introduction to dependent types in agda.
21. Louis Mandel, Cédric Pasteur, and Marc Pouzet. Time refinement in a functional synchronous language. *Sci. Comput. Program.*, 111, 2015.
22. P. Martin-Löf. *Intuitionistic type theory.*
23. P. Martin-Löf. Intuitionistic type theory. notes by giovanni sambin.
24. Jan Mikac and Paul Caspi. Temporal refinement for lustre. In *Proc. of the 5^{th} Intl. Workshop on Synchronous Languages, Applications and Programs, Edimburg, April 2005*, 2005.
25. Carroll Morgan. The refinement calculus. In *Program Design Calculi, Proc. of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992.*, 1992.
26. Ulf Norell. 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*, 2009.
27. Christine Paulin-Mohring. Modelisation of timed automata in coq. In *Theoretical Aspects of Computer Software, 4th Intl. Symp., TACS 2001, Sendai, Japan, October 29-31, 2001, Proc.*, 2001.