Clock Constraints Specification Langage : A mechanized denotational semantics in Agda

Mathieu Montin, Marc Pantel

Clock Constraints Specification Langage : A mechanized denotational semantics in Agda.

Mathieu Montin*‡, Marc Pantel*‡

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

***Abstract*—CCSL is a concurrency modeling language defined inside the MARTE UML profile. It was designed to provide system designers with requirement modeling facilities that are easier to master than temporal logics. It allows the expression of event occurrences and constraints between them. TimeSquare is a tool that implements CCSL providing on the one hand textual and graphical modeling facilities, and on the other hand a solver that checks the satisfaction of these constraints, hence giving CCSL an operational semantics. This toolset allows to assess system specific properties at the model level. But, some properties need to best verified at language level, especially when extending the language itself. In order to prove these ones, the designers of CCSL provided a denotational semantics that associate mathematical definitions to CCSL constructs. But, since this semantics is hand-made and lacks a formal version to build and verify the properties of the extensions, we propose to mechanize it in a proof assistant. We have selected the Agda theorem prover, developed by Ulf Norell at Chalmer's university for this work.**

***Index Terms*—CCSL, Denotational semantics, Proof assistant, Agda.**

I. Introduction

As real systems are getting more and more complex, a strong separation between the various concerns of the same system has become a major requirement. Specialists of each engineering domain will define their parts of the system in their own language and these parts are then integrated. The main drawback lies in the fact that many properties are not preserved during the integration of the various concerns. Thus, if you prove that the different subsystems satisfy some requirements, there is no guarantee whatsoever that the whole system will also satisfy these requirements.

A common successfull approach to tackle this problem is to abstract the different parts in the same language according to the concern handled by this language, and then to reason over this language instead of reasoning over the different sub-languages in which the subsystems have been defined. Such a high level language has to be formally specified so that its use can be trusted. The Clock Constraint Specification Language (CCSL), developed by the AOSTE team from INRIA, stands as one of the best attempt to create such a user dedicated language that targets system engineers instead of formal method experts. However, it lacks a mechanized denotational semantics to formalize all the proofs of correctness at the language level, which is what we propose in this paper.

We will start by explaining the underlying concepts of CCSL, which are the instants, the strict partial orders relating these instants and their link to the notion of Time Structure. Then we will define the clocks linking the instants to the actual modeled concerns and we will move on to more advanced concepts such as relations and expressions around clocks, in order to reach constraints definitions.

This work has been done using the Agda proof assistant, developed by Ulf Norell at the Chalmers University, and the accent will be made on the choices made to fit this tool. Agda is a dependently typed programming langage suited to both programming and building theories and proofs, thanks to the Curry-Howard isomorphism. Agda comes with several features, including (but not limited to) a type checker, a termination checker and a development environment in the form of an interactive emacs mode.

II. Related work

Our proposal provides a mechanization of the semantics of CCSL in a proof assistant. As such, this approach could be reused for other concurrent langages. Such a work has already been done using different kind of formal methods, for example [1] using Higher Order Logic in Isabelle/HOL; [2] and [3] using the Calculus of Inductive Constructions in Coq, whose description can be found in [4]. The use of Agda in this development is motivated by the expressiveness of the langage coupled with its underlying unification mechanism - in other words, Agda allows, for instance, to pattern-match on the equality proof, thus unifying its operands. This provides an interactive proof experience that other tools that do not provide unification lacks: Agda, as opposed to coq, does not rely on the application of tactics to inhabit types, but gives a well-designed framework to build them in interaction with the type checker and unifier. More on Agda can be found in [5], [6] and [7]. Although they differ from these two aspects, both of these tools rely on the same underlying intuitionistic type theory, first described in [8] and clarified in [9].

The denotational semantics of CCSL on which this work is based can be found in [10]. TimeSquare, the tool developed to describe CCSL systems as well as solve constraint sets has been presented in [11]. As for CCSL itself, it was first presented in [12]. Athough our semantics aims at being the same as the paper version, it differs through the way it has

been expressed, to best suit the constraints and the possibilities offered by Agda. An example of differences is the handling of the notion of TimeStructure - see [13] - who was translated from a constructive mathematical set theory to a generic type to better match the use of a type theory.

III. Example

We will consider a simple system throughout this paper, which can be turned On and Off, and can execute a simple action when it is On, which will remain unspecified until further notice. This system can be represented by the automaton in Figure 1.

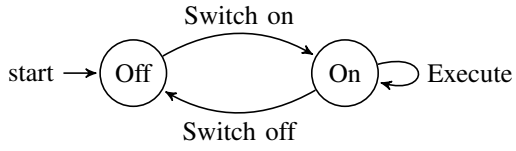

Figure 1. A simple automaton

Let's rename, for the sake of clarity, the different transitions as t_{on}, t_{off} and t_{ex}. A possible trace (i.e. sequence of transition occurences) for the system is depicted in Figure 2. This trace starts with the birth of the system and possibly continues indefinitely, which makes this finite representation partial.

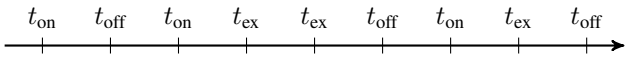

Figure 2. A possible trace

IV. Representation of time

A. Instants

Instant is the main concept on which CCSL is defined. Informally, an instant is a point in a time-line where events can occur. But this vision, which we all have instinctively, implies that two instants are always comparable, one of them being after the other one, and vice versa. However in distributed systems, there is no global clock, and one does not always know which event has occurred, or will occur, before or after any other given event, making this a poor choice to represent instants in the global case. Thus, in our framework, instants will be an unspecified Agda type - which we will call *Support* (*Instant* is the name for the algebraic structure). The cardinality of this type is unspecified as well and will remain so throughout the paper. It is important to note that `Set` in Agda represents a type, and not a mathematical set. In dependently typed langages, sets can be emulated as unary relations, as we will see later, but are not native constructs of the langages. However, by langage abuse, we will often call an Agda `Set` a set instead of a type.

```
Support : Set
```

B. Strict partial orders

Since time cannot be seen as a line on which events occurs, instants cannot be linked by a total order. Two events are not necessarily comparable, which leads to the use of partial orders to represent the possible links. In CCSL, each pair of instants is either :

- *strictly comparable*, through a precedence relation $\_\prec\_$ which means that one precedes the other one.
- *equivalent*, through a coincidence relation $\_\approx\_$ which means that, regarding our point of view, the instants cannot be differentiated.
- *independent*, which means that they are not linked by any of the two previously defined relations.

A partial order requires these relations to have 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 _≈_
record StrictPartialOrder c ℓ₁ ℓ₂ :
  Set (suc (c ⊔ ℓ₁ ⊔ ℓ₂)) where
  field
    Carrier : Set c
    _≈_     : Rel Carrier ℓ₁
    _<_     : Rel Carrier ℓ₂
    ispo    : IsStrictPartialOrder _≈_ _<_
```

This code defines two records, both dependent - a record is called dependent when one or more of its fields are typed depending on the value of other fields. Their signature is quite complicated because they are expressed at a generic level of universe (represented by the quantities a, ℓ_1 and ℓ_2 and the operators $\sqcup$ and suc) for the set and the two relations, even though they represent a simple algrebraic structure. The stratification of the universe as levels is used in type theory to reject paradoxes inherent to classical set theories.

The first record, named `IsStrictPartialOrder`, is parametrized by two relations ($\_\approx\_$ and $\_<\_$) and a set A, and states what it means for these relations to define a strict partial order over the set. It consists in four predicates which have been detailed before. The second record encapsulate this definition inside an actual strict partial order, with the relations and the set being part of the record.

C. Intervals

Since the relations that will be defined later on have a lifetime, it is useful to have the notion of intervals. Let us consider the following definition :

```
data Interval : Set where
  [_,_] : (α : Support) → (β : Support) → Interval
  [_,∞] : (α : Support) → Interval
```

We define two kinds of intervals, one that is bounded on both sides, and one that is only bounded on the left side. This asymmetry is inherent to the underlying asymmetry in a system life : it has been started at a specific time, but might be running forever. This allows us to define the belonging to an interval :

```
_∈_ : Support → Interval → Set
v ∈ [ α ,∞[ = α ≼ v
v ∈ [ α , β ] = α ≼ v × ¬ (β ≺ v)
```

These axioms entail interval inclusion :

```
_⊂_ : Interval → Interval → Set
I ⊂ J = ∀ {x} → x ∈ I → x ∈ J
```

V. Clocks

A. Intuitive definition

A clock is an entity that represents the occurrences of a specific event in a specific system. Thus, the execution of such a system is traced by the occurrences of the clocks it provides. Each of these clocks is said to tick whenever (i.e. at every instant) the event it represents occurs. In the example of Figure 1, this system exhibits three different clocks representing the occurrences of the transitions :

- "c_{on}" ticks when the system is turned on.
- "c_{off}" ticks when the system is shut down.
- "c_{ex}" ticks when the action is executed.

A possible trace for these clocks is depicted in Figure 3.

Figure 3. Ordered trace of clocks

Note that the set of instants on which a specific clock ticks must be totally ordered, which is why we could represent them on a well-defined time-line.

In this example, we represented the instants "globally" ordered as they were in Figure 2, even though this disposition actually requires constraints to be added to the system. The right representation, which is actually pointless alone, is given in Figure 4.

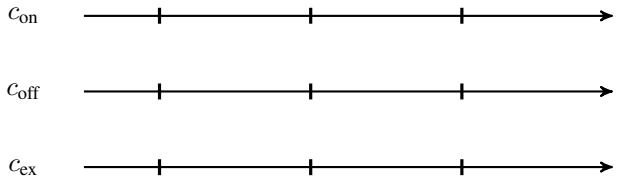

Figure 4. Unordered trace of clocks

B. Formal definition

Formally, a clock is a dependent record which define a subset of instants (the ones on which it ticks) and ensures that these instants are totally ordered. This gives the following definition :

```
record Clock : Set₁ where
  constructor
    [_o_]
  field
    Ticks  : Support → Set
    TicTot : ∀ {x y} →
      Ticks x → Ticks y → x ≠ y → x ≺ y ⊎ y ≺ x
```

A clock is a dependent record (which is implicitly parametrized by the set of instants through the module definition) which provides a constructor to build a clock and two fields. The first field is a predicate on the instants to encode the subset on which the clock ticks, and the second is the proof that the ticks of the clock are totally ordered.

C. Birth and Death

In the current state of CCSL, and in our current work accordingly, a clock contains a lot more fields. They are related to the existence of a birth and a death to encode its lifetime. In this paper however, we will not present this part of our work, since it would hide the fundamental aspects of the work behind technical difficulties.

D. Filter

It is possible to reduce the ticks of a clock regarding a certain predicate on instants. This filtering is done as follow :

```
filterBy : Clock → (Support → Set) → Clock
filterBy [ Ticks o TicTot ] P =
  [ (λ x → Ticks c x × P x)
  o (λ x₁ x₂ → TicTot c (proj₁ x₁) (proj₁ x₂)) ]
```

The function `filterBy` builds a new clock from an input clock and a predicate. The new clock will tick when both the input clock ticks and the predicates holds on a specific instant. The proof that the instants of the new clock are totally ordered is built from the proof that the input clock was totally ordered itself. Removing some of the instants does not change this property.

This definition will be particularly useful when considering some intervals as relation lifetimes. The filtering on intervals becomes :

```
_|ᵢ_ : Clock → Interval → Clock
c |ᵢ I = filterBy c (\x → x ∈ I)
```

In this case, the filtering predicate is instantiated by the membership of an instant inside a specific interval, thus reducing the ticks of the input clock to this interval.

VI. Relations

A. Definition

As shown on figure 4, clocks on their own do not give any information on the system from which they are extracted. Individually, they are shown as - potentially - endless streams of totally ordered ticks which cannot be precisely placed on

the system time-line. To overcome this limitation, a relevant system must be described by at least two clocks, and the links between them must be specified. These links can be from different types and are known as ”relations”.

A relation holds, by default, for the lifetime of the system. In Agda, we can state what it means for two clocks to be related. The global Agda type for relations is the following :

```
GlobalRelation : Set₁
GlobalRelation = Clock → Clock → Set
```

However, it is often useful to reduce the lifetime of a relation to a certain predicate over the instants (usually intervals), in which case the type becomes :

```
PredRelation : Set₁
PredRelation = Clock → Clock → (Support → Set) → Set
InteRelation : Set₁
InteRelation = Clock → Clock → Interval → Set
```

Considering a given interval, two clocks are related if they satisfy the relation whenever they tick inside this interval. In our framework, we adopted a more suitable vision of relations restricted to an interval. Two clocks are considered related on a given interval when their filtered versions are related globally through the same relation. Thus, by restricting the ”Ticks” predicate on our clock, we obtain a more natural version of interval constrained relations. Let us consider a relation ”_R_”, defined in Agda as follows :

```
postulate _R_ : GlobalRelation
```

Restraining this relation over a specific predicate (subset) is then done very easily :

```
_R_|_ : PredicatedRelation
c₁ R c₂ | P = (filterBy c₁ P) R (filterBy c₂ P)
_R_|ᵢ_ : IntervalledRelation
c₁ R c₂ |ᵢ P = (c₁ |ᵢ I) R (c₂ |ᵢ I)
```

B. Main relations

In the following section, we present some of the main relations provided by CCSL, and their representation in our framework. Since we explained how we can easily transform a global relation into a predicated relation, we will only describe these relations globally.

1) Strict precedence: A clock c_1 is said to strictly precede a clock c_2 when each consecutive ticks of c_2 is strictly preceded by a distinct and consecutive ticks of c_1. Note that the ”consecutive” word can only refer to discrete clocks. In dense clocks, the equivalent is that every ticks of c_1 placed between two mapped ticks must be mapped as well. Before getting to the formal definition of this relation, let us consider some graphical examples in Figures 5, 6 and 7.

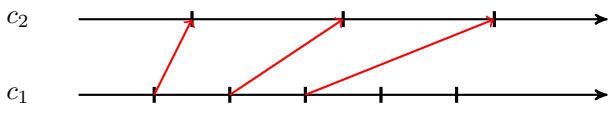

Figure 5. A standard precedence example

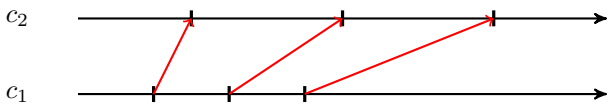

Figure 6. A specific precedence example

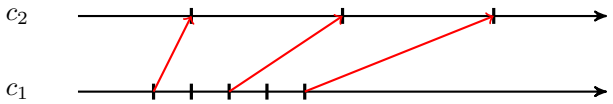

Figure 7. An incorrect precedence example

Figure 5 is the example that comes to mind when we think of this definition. Each instant of c_2 is mapped to an instant of c_1 in a way the precedence relation looks obvious. However, this definition does not force this mapping to be bijective, which means c_1 could have additional ticks that are not mapped to ticks of c_2. If these ticks occur after the ones mapped to c_2, like on figure 6, the precedence is still well-formed, as opposed to figure 7 where they are placed in between mapped ticks, thus compromising the relation. One can observe that this problem could be avoided by changing the mapped instant such that the additional ticks are always positioned as on figure 5. This seem obvious when the number of ticks is finite, yet not so much when it is not. This could be the subject of a future work.

The precedence relation requires the existence of a function which maps the instants of c_2 with the corresponding instants of c_1. It is defined as follows :

```
_|_≪′_ : (_ → _) → GlobalRelation
h | c₁ ≪′ c₂ =
  let Tc₁ = Ticks c₁ in
  let Tc₂ = Ticks c₂ in
    (∀ {i} → Tc₂ i → (Tc₁ (h i) × h i ≺ i))
    × (∀ {i j} → Tc₂ i → Tc₂ j → i ≺ j → h i ≺ h j)
    × (∀ {i j p} → Tc₂ i → Tc₂ j → p ∈ [ h i , h j ]
      → ∃ λ k → Tc₂ k × h k ≡ p)
```

```
_≪_ : GlobalRelation
c₁ ≪ c₂ = ∃ λ h → h | c₁ ≪′ c₂
```

It is transitive, and such a transitivity has been proven in the framework. The mapping function is the composition of the two underlying mapping functions. The proof is partly depicted here :

```
trans≪′ : ∀ {c₁ c₂ c₃ h₁ h₂} → c₁ ≪′ c₂ | h₁
  → c₂ ≪′ c₃ | h₂ → c₁ ≪′ c₃ | (h₁ ∘ h₂)
trans≪′ _ _ _ = ...
trans≪ : ∀ {c₁ c₂ c₃} → c₁ ≪ c₂ → c₂ ≪ c₃ → c₁ ≪ c₃
trans≪ {c₁} {c₂} {c₃} (h₁ , p₁) (h₂ , p₂) =
  h₁ ∘ h₂ , trans≪′ {c₁} {c₂} {c₃} {h₁} {h₂} p₁ p₂
```

In our example, there is a precedence relation between c_{on} and c_{off}, which leads to the timeline in Figure 8.

2) Non-strict precedence: The non-strict precedence allows two mapped instants to be coincident, thus the underlying relation is _≼_ instead of _≺_. This relation is mostly similar to the strict precedence and will not be detailed thoroughly. A simple example is however given in Figure 9.

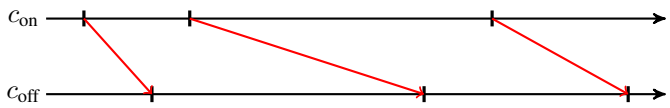

Figure 8. c_{on} precedes c_{off}

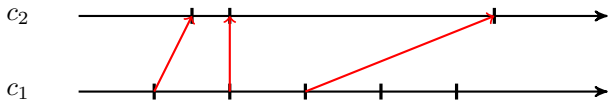

Figure 9. An example of non-strict precedence

The Agda definition is the same as the strict precedence, except for the substitution of the strict relation by the non-strict one. This relation is transitive as well, and this has been proven accordingly. The two proofs are factorized through the abstraction of the underlying relation.

3) Subclocking: A clock c_1 is said to be a subclock of a clock c_2 when every ticks of c_1 is coincident to a tick of c_2. It means that whenever c_1 ticks, c_2 ticks as well. Figure 10 shows an example of subclocking.

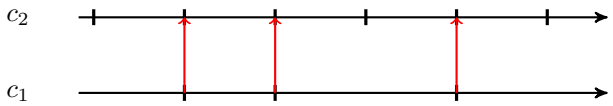

Figure 10. c_1 is a subclock of c_2

The Agda definition of this relation is fairly simple and corresponds to the intuitive one :

```
_⊑_ : GlobalRelation
c₁ ⊑ c₂ =
  let Tc₁  = Ticks c₁ in
  let Tc₂  = Ticks c₂ in
    ∀ {x₁} → Tc₁ x₁ → ∃ λ x₂ → x₁ ≈ x₂ × Tc₂ x₂
```

As for the precedence, this relation is transitive, which can be proven this way :

```
trans⊑ : ∀ {c₁ c₂ c₃} → c₁ ⊑ c₂ → c₂ ⊑ c₃ → c₁ ⊑ c₃
trans⊑ c₁c₂ c₂c₃ = λ x → let (_ , pi , qi) = (c₁c₂ x) in
  let (j , pj , qj) = (c₂c₃ qi) in j , trans≈ pi pj , qj
```

Until now, the action executed by the system while running remained unspecified. We are going to specify it to exhibit an example of subclocking. Imagine that our system is connected to a light through the use of a memory containing a variable x. This variable will be accessed and assigned by our system. The automaton in Figure 1 becomes the one in Figure 11.

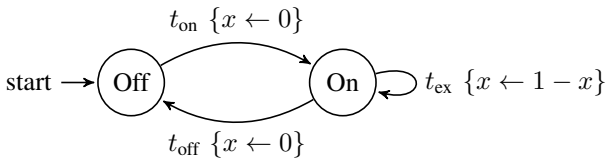

Figure 11. The automaton with a memory

The system pilots the lights through the assignement of x to 0 (light off) or 1 (light on). When the system is turned on, the light remains off until a button is pressed which turns it on. Pressing the same button will alternatively switch off the light and turn it on. Shutting down the system turns it off. In this situation, two additional clocks must be added to the system. One will trace the assignment of x to 0 and the other its assignment to 1. The resulting trace is shown in Figure 12.

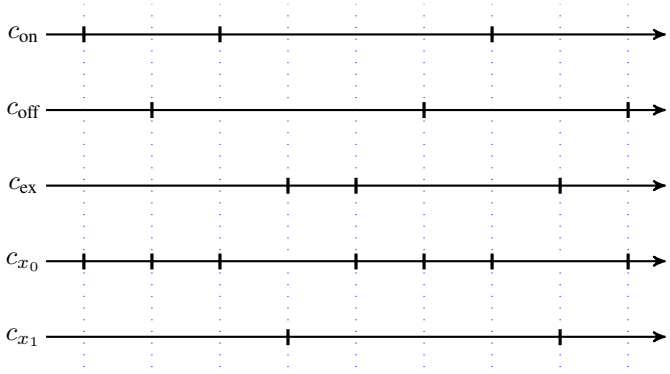

Figure 12. The trace of the system with the addition of the variable x

In this example, we can identify at least two cases of subclocking. For instance, c_{on} is a subclock of c_{x_0} and c_{x_1} is a subclock of c_{ex}.

4) Alternation: In the section about strict precedence, we stated that the clock c_{on} precedes the clock c_{off} in our example system. However, we purposely omitted to state that this precedence has to be constrained even more. Let us consider the situation in Figure 13.

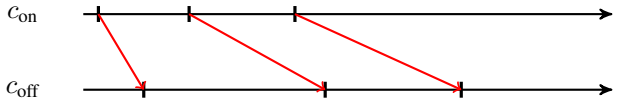

Figure 13. The precedence is true but not enough

The system is turned on twice before being shut down, which should not be possible. In some cases, the precedence itself is not strong enough to specify the behavior of two clocks. The need to express the alternation between events comes from this original lack. Two clocks are said to be alternated when one precedes the other in such a way that two ticks of a clock cannot occur in between two ticks of the other one. Note that the underlying precedence has to be strict for the relation to be consistent. A non-strict precedence would lead to illformed cases of alternation. In this case, the trace of our system is actually the one presented on figure 8, on which we added the additional informations shown on Figure 14.

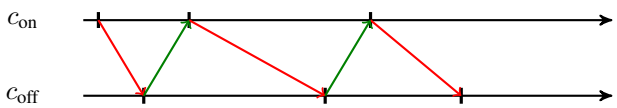

Figure 14. c_{on} alternates with c_{off}

In Agda, this relation is defined as follow :

```
_≪≫_ : GlobalRelation
c₁ ≪≫ c₂ = ∃ λ h → c₁ ≪′ c₂ | h
  × ∀ {i j} → i ≺ j → i ≺ h j
```

5) Equality: Two clocks c_1 and c_2 are equal when they only tick on coincident instant. It means that if c_1 ticks on i then there exists an instant k which coincides with i and where c_2 ticks. An example is represented in Figure 15.

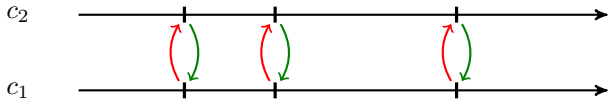

Figure 15. c_1 is equal to c_2

This definition is exactly equivalent to a double subclocking, which is how we have defined it in Agda.

```
_∽_ : GlobalRelation
c₁ ∽ c₂ = c₁ ⊑ c₂ × c₂ ⊑ c₁
```

Since the underlying relation is an equivalence we can easily prove that this new relation is an equivalence as well :

```
eq∽ : IsEquivalence _∽_
eq∽ = ...
```

6) Exclusion: Two clocks are in exclusion when they have no coincident ticks. The Agda definition for this relation is fairly simple, and we will not provide any graphical example considering the nature of the relation.

```
_♯_ : GlobalRelation
c₁ ♯ c₂ =
  let Tc₁ = Ticks c₁ in
  let Tc₂ = Ticks c₂ in
    ∀ {x y} → Tc₁ x → Tc₂ y → ¬ x ≈ y
```

VII. Expressions

A. Definitions

CCSL allows to define new clocks from existing clocks. But the formal definition of such operations can be complicated. Creating new clocks usually sets an arbitrary order between the instants on which the underlying clocks are ticking, which means that two clocks apparently independent are getting related because a new clock is created from them. The common example is the union. The union of two clocks ticks whenever one of the two clocks ticks. Since a clock has a total order on its ticks, the ticks of the union must be totally ordered, which leads to a total order on the ticks of the two other clocks. To handle this difficulty, we do not propose to create clocks in a functional way, but to relate the three clocks using predicates to state that a clock could be the result of such operations, thus deviating from the original CCSL denotational semantics.

Unlike relations, expressions will not usually be defined on a specific interval. However, it is made possible in the framework by filtering the resulting clock on the interval.

The global type of expressions is defined in our work as a relation between three clocks :

```
GlobalExpression : Set₁
GlobalExpression = Clock → Clock → Clock → Set
```

1) Intersection: A common expression on clocks is the intersection. The clocks which results from the intersection of two clocks only ticks on each instant they simultaneously tick. This expression is a predicate on the three clocks.

```
_≡_∩_ : GlobalExpression
c ≡ c₁ ∩ c₂ =
  let Tc₁ = Ticks c₁ in
  let Tc₂ = Ticks c₂ in
  let Tc = Ticks c in
    (∀ {i} → Tc i → ∃ \j → ∃ \k →
      Tc₁ j × Tc₂ k × i ≈ j × i ≈ k)
    × (∀ {i j} → Tc₁ i → Tc₂ j → i ≈ j →
      ∃ \k → Tc k × k ≈ i)
```

This first part of this predicate states that whenever c ticks on an instant i, there exists two instants j and k which are coincident to i and on which both c_1 and c_2 ticks respectively.

The second part states that if c_1 ticks on i, c_2 ticks on j, and if these instants are coincident, then c ticks on an instant coincident to them. Figure 16 shows an example of intersection.

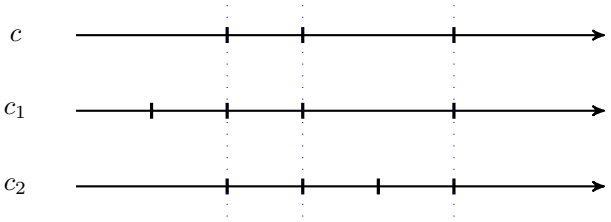

Figure 16. An example of intersection

It is important to note that this expression does not change the relations between the instants on which the underlying clocks tick. Indeed, the total order induced by the definition of c was already present in both c_1 and c_2

2) Union: As explained in the introduction of this part, the union is a tricky expression to fully understand, despite its apparent simplicity, because it induces an order on instants which was not present before. Not only does it build a new clock but it also constrains the traces of the system. The following predicate explains what it means for a clock to be the union of two other clocks, and handles this issue.

```
_≡_∪_ : Clock → Clock → Clock → Set
c ≡ c₁ ∪ c₂ =
  let Tc₁ = Ticks c₁ in
  let Tc₂ = Ticks c₂ in
  let Tc = Ticks c in
    (∀ {i} → (Tc₁ i ⊎ Tc₂ i) → ∃ \j → i ≈ j × Tc j)
    × (∀ {i} → Tc i → ∃ \j → i ≈ j × (Tc₁ j ⊎ Tc₂ j))
```

The first part of this predicate states that if either c_1 or c_2 ticks on an instant i then there exists an instant coincident to i on which c ticks.

The second part states that if c ticks on an instant i then there exists an instant j which is coincident to i and on which either c_1 or c_2 ticks. Figure 17 depicts an example of union.

Note that this example does not show how the instants are being ordered by the expression. It makes it look like they already are, which is not the case. The clock c happens to be consistent with the idea of the union of c_1 and c_2, but it is not the result of any operation.

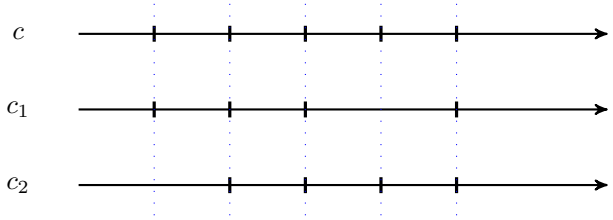

Figure 17. An example of union

3) Other expressions: There exists a lot of other expressions (either fundamental or derivate), some of them depending on the death instant, some other being induced by a natural number. None of them will be detailed in this paper, whose goal is not to present all CCSL constructs, but to explain the ideas behind their mechanization.

VIII. Properties

A. CCSL specification

In this work, instants are represented as a classic set with an unspecified strict partial order relation. Every construct from CCSL which we specified in Agda is expressed using this set, which is passed as a parameter to the different modules. This view is different from the one of the CCSL creators, who see the instants of a system (the Time Structure) as the union of all the instants on which its different clocks ticks. This vision that synthesize the *Support* set from the clocks, is unsuitable for tools like Agda. Indeed, as we saw when defining the clocks, sets are not native in this kind of langages and are emulated by predicates. They are not actual sets as seen in the ZFC theory. This is why we had to change the status of the instants and the Time Structure they form. Providing a specific set as instants and a specific relation to order them would mean solving the ordering of the events induced by the clocks of a system and the relations / expressions defined around them.

A CCSL specification is a set of constraints applied to a set of clocks. These constraints can be either relations or expressions, since both of these can influence the underlying ordering of the instants. The goal of this work is not to solve a set of constraints (this is done by the TimeSquare tool) but to provide a mechanised semantics for CCSL. It can be used to define and validate additions to the language that may remain unclear or unspecified in a paper version. One of these additions is the instant refinement which will be presented in a further section. Regarding a CCSL specification, one of the goals of our work is to reduce the set of constraints it contains. For instance, if one of the constraints in the set can be deduced from the other one, it should be removed. Another example is if one of the clocks needs to be hidden from the specification, all constraints linked to it must disappear without any loss of information regarding the other clocks. In both cases, we need properties linking the different constraints in order to achieve some unifications between them.

The following section presents some of the properties we proved in our framework. Most of these properties are fairly easy to understand, but the proofs are not necessarily simple, and will not be fully detailed. The transitivity properties have already been mentioned and will be leftout of this section.

B. Example of properties

Let c_0, c, c_1, c_2 and c_3 be five clocks.

1) Subclock and exclusion: If c_1 is in exclusion of c_3 and if c_2 is a subclock of c_3 then c_1 is in exclusion of c_2 as well. This is intuitive since c_2 ticks at most each time c_3 ticks. This can be expressed and proven in Agda :

```
excluSub : ∀ {c₁ c₂ c₃} → c₁ ♯ c₃ → c₂ ⊑ c₃ → c₁ ♯ c₂
excluSub c₁♯c₃ c₂⊑c₃ t₁x t₂y with c₂⊑c₃ t₂y
excluSub c₁♯c₃ c₂⊑c₃ t₁x t₂y | a , y≈a , ta
  = λ x → c₁♯c₃ t₁x ta (trans≈ x y≈a)
```

2) The union is symmetrical: If c can be viewed as the union of c_1 and c_2 then it can also be viewed as the union of c_2 and c_3.

To prove this property, we need to be able to swap a sum of types, which is done by the following function.

```
flipSum : ∀ {a b} {A : Set a} {B : Set b} →
  A ⊎ B → B ⊎ A
flipSum (inj₁ x) = inj₂ x
flipSum (inj₂ y) = inj₁ y
```

Proving that the union is symmetrical is done as follows :

```
symUnion : ∀ {c c₁ c₂} → c ≡ c₁ ∪ c₂ → c ≡ c₂ ∪ c₁
symUnion (proj₁ , proj₂ , proj₃) =
  (λ x → proj₁ (flipSum x)) ,
  (λ x → let (a , b , c) = proj₂ x in a , b , flipSum c) ,
  flipSum proj₃
```

3) Union and subclocking: We can prove that each component of a union is a subclock of the union. That can be proved both ways (for both clocks) using the symmetry of the union.

```
subUnionₗ : ∀ {c c₁ c₂} → c ≡ c₁ ∪ c₂ → c₁ ⊑ c
subUnionₗ (p₁ , _ , _) = λ x → p₁ (inj₁ x)

subUnionᵣ : ∀ {c c₁ c₂} → c ≡ c₁ ∪ c₂ → c₂ ⊑ c
subUnionᵣ {c} {c₁} {c₂} p =
  subUnionₗ {c} {c₂} {c₁} (symUnion {c} {c₁} {c₂} p)
```

4) Unicity of union: We can prove the unicity of the union relatively to the equivalence classes induced by the underlying equivalence relation. We start by proving that if two clocks correspond to the same union, one is a subclock of the other.

```
uu : ∀ {c₀ c c₁ c₂} →
  c₀ ≡ c₁ ∪ c₂ → c ≡ c₁ ∪ c₂ → c ⊑ c₀
uu (proj₁ , proj₂ , proj₃) (proj₄ , proj₅ , proj₆) =
  λ x → let (_ , ≈x , tx) = proj₅ x in
        let (y , ≈y , ty) = proj₁ tx in
          y , trans≈ ≈x ≈y , ty
```

We conclude by applying the previous property both ways.

```
unicityUnion : ∀ {c₀ c c₁ c₂} →
  c₀ ≡ c₁ ∪ c₂ → c ≡ c₁ ∪ c₂ → c ∽ c₀
unicityUnion {c₀} {c} {c₁} {c₂} p q =
  uu {c₀} {c} {c₁} {c₂} p q , uu {c} {c₀} {c₁} {c₂} q p
```

5) The intersection is symmetrical: It is possible to prove that the intersection is also symmetrical. The proof is a bit more verbose than the one for the union, but not quite complicated as well.

```
symInter : ∀ {c c₁ c₂} → c ≡ c₁ ∩ c₂ → c ≡ c₂ ∩ c₁
symInter (proj₁ , proj₂ , proj₃ , proj₄) =
  (λ ti →
    let (j , k , tj , tk , i≈j , i≈k) = proj₁ ti in
      k , j , tk , tj , i≈k , i≈j) ,
  (λ ti tj i≈j →
    let (k , tk , k≈i) = proj₂ tj ti (sym≈ i≈j) in
      k , tk , trans≈ k≈i (sym≈ i≈j)) ,
  (proj₄ , proj₃)
```

6) Intersection and subclocking: As opposed to the union, when c is enforced to be the intersection of c_1 and c_2, c is a subclock of both of them, which can be proven.

```
subInterₗ : ∀ {c c₁ c₂} → c ≡ c₁ ∩ c₂ → c ⊑ c₁
subInterₗ (p , _ , _) = λ x →
  let (j , _ , tj , _ , i≈j , _) = p x in
    j , i≈j , tj
```

```
subInterᵣ : ∀ {c c₁ c₂} → c ≡ c₁ ∩ c₂ → c ⊑ c₂
subInterᵣ {c} {c₁} {c₂} c≡c₁∩c₂ =
  subInterₗ {c} {c₂} {c₁} (symInter {c} {c₁} {c₂} c≡c₁∩c₂)
```

7) Unicity of intersection: As for the union, we can prove that the intersection is unique.

```
ui : ∀ {c₀ c c₁ c₂} →
  c₀ ≡ c₁ ∩ c₂ → c ≡ c₁ ∩ c₂ → c ⊑ c₀
ui (proj₁ , proj₂ , proj₃) (proj₄ , proj₅ , proj₆) =
  λ x →
    let (j , k , tj , tk , i≈j , i≈k) = proj₄ x in
    let (l , tl , l≈j) =
      proj₂ tj tk (trans≈ (sym≈ i≈j) i≈k) in
        l , trans≈ i≈j (sym≈ l≈j) , tl

unicityInter : ∀ {c₀ c c₁ c₂} →
  c₀ ≡ c₁ ∩ c₂ → c ≡ c₁ ∩ c₂ → c ∽ c₀
unicityInter {c₀} {c} {c₁} {c₂} p q =
  ui {c₀} {c} {c₁} {c₂} p q , ui {c} {c₀} {c₁} {c₂} q p
```

8) Intersection and union: As a consequence of these proofs, we can easily prove that the intersection is a subclock of the union, using the transitivity of the subclocking.

```
subInterUnion : ∀ {c₀ c c₁ c₂} →
  c₀ ≡ c₁ ∩ c₂ → c ≡ c₁ ∪ c₂ → c₀ ⊑ c
subInterUnion {c₀} {c} {c₁} {c₂} c₀≡c₁∩c₂ c≡c₁∪c₂ =
  trans⊑′ {c₀} {c₁} {c}
    (subInterₗ {c₀} {c₁} {c₂} c₀≡c₁∩c₂)
    (subUnionₗ {c} {c₁} {c₂} c≡c₁∪c₂)
```

9) A clock Lattice: With these proofs and some others on intersection and union, the set of clocks exhibits the mathematical structure of a lattice, with the sup operator being the union, and the inf operator being the intersection.

IX. Conclusion

A. Summary

In this work, we have proposed a mechanization of CCSL in Agda. We have clarified certain notions inherent to this language, and have proposed ways of encoding it in a proof assistant. As stated in the first part of this paper, details about the lifetime of a clock, encoded as a birth instant and a death instant have been omited. Their presentation would not have been suitable to this article. However, this has been encoded in the framework and will be presented later on. This work stands as an example of mechanisation in Agda for a concurrent langage, as well as an attempt to provide the CCSL developpers with a complete mechanized semantics from which different features could eventually be extracted, as explained in the next section.

B. Future work

This work brings different perspectives that would complete and extend both CCSL and our semantics :

- We will define and prove as many properties as possible over the relations and the expressions defined in CCSL, in order to provide a correct way to reduce the set of constraints related to a certain specification. This will be done by computing derived constraints and comparing them to those that have been provided in the set.
- We will extend the langage through the definition of instant refinement in order to encode the notions of simulation, bisimulation and weak bisimulation in the framework to get a better hold over them. It requires to consider sets of clocks and the relations that bind them.
- We will go deeper into the definition of the birth and death instants to handle some difficulties that emerge with these notions. For instance, they induce loss of some algebraic properties which we would like to handle properly.

Acknowledgment

The authors would like to thank Julien DeAntoni, Robert De Simone and Frédéric Mallet for providing us with their time and valuable expertise regarding CCSL. This work could not have been done without them.

References

[1] R. Hale, R. Cardell-Oliver, and J. Herbert, "An embedding of timed transition systems in HOL," *Formal Methods in System Design*, vol. 3, no. 1/2, 1993.

[2] M. Garnacho, J. Bodeveix, and M. Filali-Amine, "A mechanized semantic framework for real-time systems," in *Formal Modeling and Analysis of Timed Systems - 11th International Conference, FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proceedings*, 2013.

[3] C. Paulin-Mohring, "Modelisation of timed automata in coq," in *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings*, 2001.

[4] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series, 2004.

[5] U. Norell, "Dependently typed programming in agda," in *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, 2009.

[6] J. Malakhovski, "Brutal [meta]introduction to dependent types in agda."

[7] A. Bove and P. Dybjer, "Dependent types at work," in *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, 2008, pp. 57–99.

[8] P. Martin-Löf, *Intuitionistic type theory.*

[9] ——, "Intuitionistic type theory. notes by giovanni sambin."

[10] J. Deantoni, C. André, and R. Gascon, "CCSL denotational semantics," Research Report RR-8628, 2014.

[11] J. Deantoni and F. Mallet, "TimeSquare: Treat your Models with Logical Time," in *TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012*, 2012.

[12] C. André and F. Mallet, "Clock Constraints in UML/MARTE CCSL," INRIA, Research Report RR-6540, 2008.

[13] G. Winskel, "Event structures," in *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, 1986.