Mechanizing the denotational semantics of the Clock Constraint Specification Language

Mathieu Montin[1,2,3] and Marc Pantel[1,2,4]

[1] Université de Toulouse ; Toulouse INP, *IRIT*
[2] *CNRS* ; Institut de Recherche en Informatique de Toulouse (IRIT)
[3] mathieu.montin@enseeiht.fr, http://montin.perso.enseeiht.fr
[4] marc.pantel@enseeiht.fr, http://pantel.perso.enseeiht.fr

Abstract. Domain Specific Modelling Languages provide the designers with appropriate languages for the task they must conduct. These dedicated languages play a key role in popular Model Driven Engineering (MDE) approaches. Their semantics are usually written in a semi-formal manner mixing natural language and mathematical notations. The mechanization of these semantics rely on formal specification languages. They are usually conducted in order to assess the correctness of verification and transformation tools for such languages. This contribution illustrates such a mechanization for the Clock Constraint Specification Language (CCSL). This language allows to model the timed concurrency concern in the MARTE UML profile and was designed to be easier to master than temporal logics for the system engineers. Its semantics has been defined in the usual semi-formal manner and implemented in the TimeSquare simulation tool. We discuss the interest of this mechanization and show how it allowed to prove properties about this language and ease the definition of a refinement relation for such models. This work relies on the Agda proof assistant and is presented accordingly.

Keywords: DSML · Semantics mechanization · Proof assistants · CCSL

1 Introduction

As systems are getting more and more complex, a strong separation between the various concerns in a system has become a major requirement. Specialists of each engineering domain define their views of the system in their own language, called a Domain Specific Modelling Language (DSML) and these views are then integrated. There are two main drawbacks of this approach: first, these languages, and especially their semantics, are usually defined in a semi-formal way, thus complicating their common understanding; and second, many properties are not preserved during the integration of the various parts as the same concerns are expressed in different DSMLs. Thus, if each concern in different views satisfies some requirements, there is no guarantee the concern of the whole system combining these views will satisfy the same requirements.

A promising approach to tackle this problem is to abstract the common concerns from the various parts expressed in different DSMLs in a common

DSML. It allows to reason over this single DSML instead of the different DSMLs of the various views. Then, the whole semantics of this common DSML should be defined in a formal manner to provide a formal semantics for the concern in each DSML. The Clock Constraint Specification Language (CCSL) from the UML MARTE standard, developed by the AOSTE team from INRIA, provides such a user dedicated language for the concurrency concern. Its operational semantics is defined as an interpreter in the TimeSquare tool-set, but it lacks a mechanized denotational semantics to formalize its underlying concepts and conduct proofs both on models and associated tools, which is the core purpose of this work. An added value of this mechanization is that it allowed to detect several issues in the semi-formal CCSL denotational semantics.

This papers starts by defining core aspects of CCSL: The instants which represent the event occurrences, the strict partial orders binding these instants together, the clocks which are entities linking the instants to the actual modelled concerns and finally more advanced concepts such as relations and expressions around clocks to reach constraints definitions.

This work has been done using the Agda proof assistant (a language and tool-set developed by Ulf Norell). Since a semi-formal denotational semantics of CCSL already exists, the accent will be made throughout this paper on the choices made to fit Agda. Pieces of Agda code will be given to point out different aspects in our mechanization. They depict either data structures, definitions or properties. Their representation in this paper is partial and some details have been omitted to make them more understandable. These omissions include some levels of universe, the explicit substitution for some implicit parameters as well as some operators used to adapt the types of some terms. This last family of hidden details is useful in the actual development because the relations and functions are defined on instants where clocks tick, which are represented as pairs of values (the witness instant and the proof that the clock ticks on that instant, encoded as $\exists$ types). The whole development is available on the first author's web page.

2 Representation of time

2.1 Instants

The main underlying concept of CCSL is the instants. Informally, an instant is a point in a time-line where events can occur (dually a time-line is a sequence of strictly ordered instants). It fits the common understanding of events that occur at a specific time and can be preceded or followed by occurrences of other events. This vision of time is usually modelled using numbers (real numbers or integers) to represent such instants because they are totally ordered. In distributed systems however, this vision is usually unsuitable because these total orders, however existent, cannot be observed. Only a partial representation of time can be specified and leads to the use of partial order to model time, which will be briefly described later on. This means that the use of numbers to represent time is not any more relevant than any other abstract set. In operational

semantics, they will be used again because a specific linearisation of time is chosen but this is not the case in our work. For this reason, instants in our work is an unspecified Agda type - named *Support*, while *Instant* is the name of the algebraic structure it forms when coupled with a partial order.

2.2 Strict partial orders

The common vision of a unique time-line on which events occur implies that two instants are always comparable precedence-wise (like numbers with their common order relation). However, in distributed systems, there is no global clock, and only some events can be compared to each other. Partial orders are thus used to represent the possible relations between the instants. In CCSL, each pair of instants is either *strictly comparable*, through a precedence relation $\prec$, *equivalent*, through a coincidence relation $\approx$ or neither of them.

2.3 CCSL specification

In this work, instants are represented as a classic set with an unspecified strict partial order relation. Every CCSL construct specified in Agda is expressed using this set, which is passed as a parameter to the different modules. This view is different from the CCSL creators' one, who see the instants of a system (the Time Structure[15]) as the union of all the instants on which the different clocks tick. This vision, synthesizing the *Support* set from the clocks, is not suitable for both denotational semantics and tools like Agda. Indeed, sets are not axiomatic in Agda and are emulated by predicates which are not usual sets as seen in the ZFC theory. Thus, we had to change the status of the instants and the associated Time Structure. This vision is more abstract and more suitable to building generic proofs. It is then possible to assess if a given operational semantics behaves as an instance of this more formal semantics.

3 Clocks

3.1 Intuitive definition

A clock is an entity that tracks the occurrences of a specific event in a given system. A clock ticks whenever (i.e. at every instant) the event it represents occurs. Such a system is represented by a set of clocks representing any possible event that can occur during its execution. Each clock usually ticks an infinite number of times – can be both $\aleph_0$ (countable clocks representing discrete or dense time) or $\aleph_1$ (uncountable clocks representing only dense time)– and is partially represented in a time-line such as Figure 1. Discrete time means that, between two ordered instants, there exists only a finite number of other instants. Dense time means that, between two ordered instants, there exists always an infinite number of other instants. In this example, the clock called `c` ticks three times during the portion of time depicted in the diagram. The ticks are separated by a certain

amount of time, unspecified – there is no scale on the diagram – because such a system is usually asynchronous. Thus, the only relevant information depicted in this diagram is that the event tracked by `c` occurred at least three times throughout the lifetime of the system. This is however a very poor information which must be completed with the addition of other clocks and constraints between them.

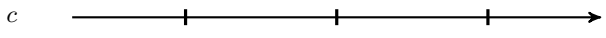

Fig. 1. An example of a clock c

3.2 Formal definition

Formally, a clock is an Agda record which contains a subset of instants (the ones on which it ticks) and the proof that these instants are totally ordered:

```
record Clock : Set where
  constructor
    [_∘_]
  field
    Ticks  : Pred Support
    TicTot : IsStrictTotalOrder {A = ∃ Ticks} (_≡_ on proj₁) (_≺_ on proj₁)
```

This clock record provides a constructor – `[_∘_]` – to build a clock and two fields – `Ticks` and `TicTot`. The first field is a predicate (`Pred`) 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 (`IsStrictTotalOrder`). The constructor has two underscores where its parameters will be placed – Assuming `t` is a subset of instants and `tot` is the proof that the underlying partial order form a strict total order on this subset of instants, then `[ t ∘ tot ]` is a Clock. Note that the underlying set of this strict partial order is `∃ Ticks` which can be seen as the subset of instants on which the clock ticks. More technically, `∃ Ticks` is the type of elements of the form `(x , Tx)` where `x` is an instant and `Tx` the proof that the clock ticks on `x`. The underlying relations of the strict partial order are the projections of the coincidence and precedence relations – of the partial order binding the instants – on the first element of these couples. For instance, `≡ on proj₁` is defined this way : `(x , Px) ≡ (y , Py) on proj₁` $\Leftrightarrow$ `x ≡ y`.

4 Relations

4.1 Definition

In a complex and possibly heterogeneous system, many events – hence many clocks – can be identified. An important aspect of CCSL is that it does not handle differently complex and heterogeneous systems (in a way, in CCSL, each

Fig. 2. Some instants are bond

system is heterogeneous compared to the atomistic description of each event it provides). Each clock taken separately does not offer many interesting information about the whole system, but bound together, they provide useful specification about its global behaviour. This binding can be given as relations that constrain the execution of the system and, in our framework, are described as predicates over two clocks (mathematical relations). They enforce an order between some instants by requiring some of them to be bound by precedence – red arrows – or by coincidence – dashed blue lines – as depicted in Figure 2. A relation holds, by default, for the lifetime of the system. The global Agda type for relations is:

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

Any predicate over two clocks (any set of couples of clocks) is a clock relation.

4.2 Main relations

CCSL provides several relations. Some are defined on generic clocks (both dense and discrete). Others are restricted to discrete clocks. This paper only handles the first kind presented in this section. The other kind is the object of a future work.

Strict precedence A clock c_1 is said to strictly precede another 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. This mapping refers to the precedence function that binds the instants of the two clocks together. This function will be described more formally later on. Before getting to the formal definition of this relation, let us consider some examples in Figures 3, 4 and 5.

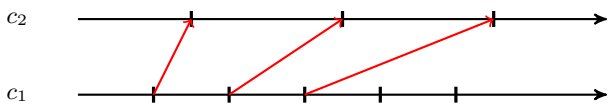

Fig. 3. A standard precedence example

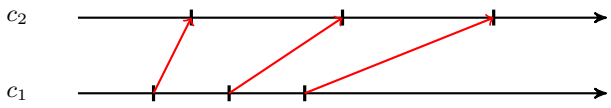

Fig. 4. A specific precedence example

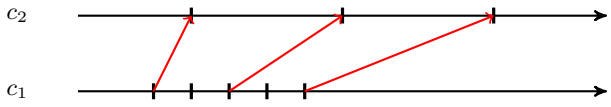

Fig. 5. An incorrect precedence example

In Figure 3, each instant of c_2 is mapped to an instant of c_1 in a way that the precedence relation looks obvious. However, this definition does not require 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 4, the precedence is still well-formed, as opposed to figure 5 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 3. This seems obvious when the number of ticks is finite, yet not so much when it is not. The current paper version of CCSL denotational semantics does make that last distinction between a well-formed and ill-formed precedence. This was an issue that our mechanization work has revealed. This will be tackled in future versions of CCSL and TimeSquare.

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

```
1  [_]_≪_ : (_ → _) → Relation
2  [ h ] [ Tc₁ ∘ _ ] ≪ [ Tc₂ ∘ _ ] = ∀ (i j : ∃ Tc₂) p →
3    (h i ∈ Tc₁ × h i ≺ i) ×
4    (i ≺ j → h i ≺ h j) ×
5    (p ∈ᵢ ⟦ h i - h j ⟧ → ∃ λ (k : ∃ Tc₂) → h k ≈ p)
```

This definition is composed of three predicates, at lines 3, 4 and 5. The first one – line 3 – ensures that all ticks of c_2 are mapped with ticks that respect the required precedence; the second one – line 4 – ensures that the binding function preserves the precedence order; the third one – line 5 – ensures that there is no unmapped instants between two mapped instants. The $\forall$ is a syntactic sugar to introduce quantities while $\times$ can be seen as the logical "and" and the brackets are delimiters for an interval. As a consequence of this definition, two clocks are related by precedence if there exists a function such that they are related through it:

```
_≪_ : Relation
c₁ ≪ c₂ = ∃ λ h → [ h ] c₁ ≪ c₂
```

The λ here is a syntactic element used to introduce a new variable `h` in the context from an existence proof ($\exists$). This definition is transitive, and such a transitivity has been proven in the framework, but is not presented here.

Non-strict precedence The non-strict precedence allows two mapped instants to be coincident, thus the underlying relation is $\_\preccurlyeq\_$ instead of $\_\prec\_$. This relation is mostly similar to the strict precedence and will not be detailed thoroughly. A simple example is given in Figure 6.

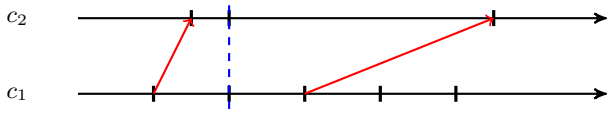

Fig. 6. An example of non-strict precedence

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 has been proven transitive as well. The two proofs are factorized through the abstraction of the underlying relation (as well as are the definitions).

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 7 shows an example of subclocking.

Fig. 7. c_1 is a subclock of c_2

The Agda definition of this relation is as follows:

```
_⊑_ : Relation
[ Tc₁ ∘ _ ] ⊑ [ Tc₂ ∘ _ ] = ∀ (x : ∃ Tc₁) → ∃ λ (y : ∃ Tc₂) → x ≈ y
```

It states that whenever c_1 ticks on an instant x_1 – `∀ (x : ∃ Tc₁)` – there exist an instant x_2 on which c_2 ticks – `∃ λ (y : ∃ Tc₂)` – which coincides with x_1 . This relation is transitive:

```
trans⊑ : ∀ {c₁ c₂ c₃} → c₁ ⊑ c₂ → c₂ ⊑ c₃ → c₁ ⊑ c₃
trans⊑ c₁c₂ _    x with c₁c₂ x
trans⊑ _    c₂c₃ _ | y , _   with c₂c₃ y
trans⊑ _    _    _ | _ , x≈y | z , y≈z = z , trans≈ x≈y y≈z
```

This proof uses a `with` construct which allows to add new quantities to the context – and usually case split on them. It relies on the transitivity of the underlying coincidence relation and combines it with the two inputs representing the subclocking proofs.

Alternation There are some cases where precedence is not enough to fully express the semantics or their relation. In Figure 8, the clock c_1 ticks a third time before the clock c_2 ticks a second time.

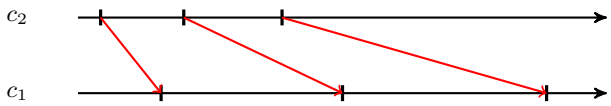

Fig. 8. The precedence is insufficient

There are some cases where this kind of behaviour might be unwanted and must be forbidden accordingly, forcing the the clocks to be further constrained. This additional constraint coupled with the original precedence is called alternation. 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 ill formed cases of alternation. In this case, the trace of our system is actually the one presented on figure 9.

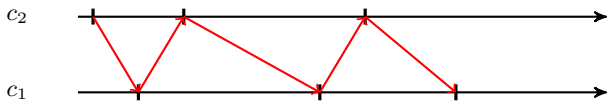

Fig. 9. c_1 alternates with c_2

In our framework, this relation is defined as follow:

```
_≪≫_ : Relation
c₁ ≪≫ c₂ = ∃ λ h → [ h ] c₁ ≪ c₂ × (∀ (x y : ∃ (Ticks c₂)) → x ≺ y → x ≺ h y)
```

c_1 alternates with c_2 when the two following predicates hold: there exists a function h such that c_1 strictly precedes c_2 through h; and h satisfies a certain predicate through the precedence relation, hence enabling the alternation instead of the simple precedence. It is thus trivial that alternation implies precedence.

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 j which coincides with i and where c_2 ticks. An example is represented in Figure 10.

Fig. 10. c_1 is equal to c_2

This definition is exactly equivalent to a double subclocking:

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

This relation has been proven to be an equivalence.

Exclusion Two clocks are in exclusion when they have no coincident ticks. An example of exclusion is given on figure 11.

Fig. 11. c_1 is in exclusion with c_2

The Agda definition is the following:

```
_♯_ : Relation
[ Tc₁ ∘ _ ] ♯ [ Tc₂ ∘ _ ] = ∀ (x : ∃ Tc₁) (y : ∃ Tc₂) → ¬ x ≈ y
```

This definition consists of a predicate that for any x and y, if c_1 ticks on x and c_2 ticks on y, then x and y are not coincident.

5 Expressions

5.1 Definition

CCSL allows the definition of new clocks from existing clocks, which is acceptable from an operational point of view. Creating new clocks usually sets an arbitrary order between the instants on which the underlying clocks are ticking, which means that instants apparently independent are getting related because a new clock is created out of 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. In our denotational framework, everything is already existing, thus we cannot create such new clocks. We assume

they already exist and propose to relate them using predicates to state that a clock could be the result of such operation. To better comprehend this notion, let us take the example of the addition between natural numbers. One can say that 3 is the result of the operation 1 + 2 while another point of view could be that the triplet (3,2,1) is a member of the addition. We take the second point of view to better match the denotational aspect of our work. The type of expressions is thus defined as a relation between three clocks:

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

5.2 Examples of expressions

Intersection A common expression on clocks is the intersection. The clocks which results from the intersection of two clocks only ticks on each instant where they simultaneously tick:

```
_≡_∩_ : Expression
[ Tc ∘ _ ] ≡ [ Tc₁ ∘ _ ] ∩ [ Tc₂ ∘ _ ] =
  (∀ (x : ∃ Tc) → ∃ λ (y : ∃ Tc₁) → ∃ λ (z : ∃ Tc₂) → x ≈ y × x ≈ z) ×
  (∀ (y : ∃ Tc₁) (z : ∃ Tc₂) → y ≈ z → ∃ λ (x : ∃ Tc) → x ≈ y)
```

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 12 shows an example of intersection.

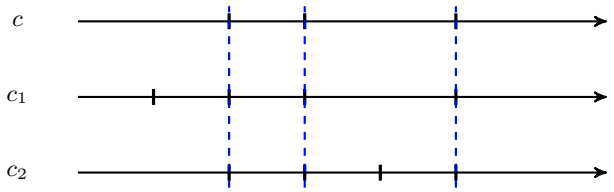

Fig. 12. An example of intersection

Union The following predicate explains what it means for a clock to be the union of two other clocks.

```
_≡_∪_ : Expression
[ Tc ∘ _ ] ≡ [ Tc₁ ∘ _ ] ∪ [ Tc₂ ∘ _ ] =
  (∀ (x : ∃ (Tc₁ ∪ Tc₂)) → ∃ λ (y : ∃ Tc) → x ≈ y) ×
  (∀ (y : ∃ Tc) → ∃ λ (x : ∃ (Tc₁ ∪ Tc₂)) → x ≈ y)
```

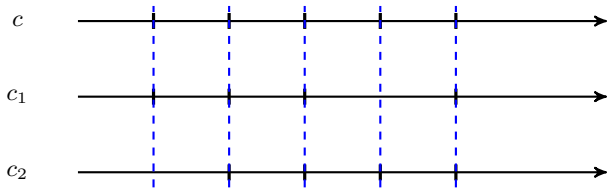

Fig. 13. An example of union

The first part of this predicate states that if either c_1 or c_2 ticks on an instant x then there exists an instant y coincident to x on which c ticks. The second part states that if c ticks on an instant y then there exists an instant x coincident to y and on which either c_1 or c_2 ticks. Figure 13 is an example of union.

Note that in our framework and example, 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.

Other expressions There exists a lot of other expressions (either fundamental or derivative), 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.

6 Properties

One advantage of mechanizing a semi-formal semantics is that this one can be validated by proving algebraic properties of the various operators, thus improving confidence in the language definition.

6.1 Goal

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 INRIA TimeSquare tool) but to provide a mechanized 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 is available at [11]. 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 relating the different constraints in order to achieve some unifications between them.

Moreover, we also need these properties to assess the correctness of our denotational semantics regarding the common behaviour one expect about clocks, relations and expressions. This section presents some of the ones we proved in our framework. Most of these properties are not conceptually challenging, but the proofs are not necessarily simple, and will not always be fully detailed. For instance, the transitivity properties have already been mentioned and will be left out of this section. It is important to understand that these properties are fundamental because they are the foundation on which more advanced use cases could be built.

6.2 Examples of properties

Subclock and exclusion If c_1 is in exclusion with c_3 and if c_2 is a subclock of c_3 then c_1 is in exclusion with 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 _     c₂⊑c₃ _ y _   with c₂⊑c₃ y
excluSub c₁♯c₃ _     x _ x≈y | z , y≈z = c₁♯c₃ x z (trans≈ x≈y y≈z)
```

The union is commutative 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_1. 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
```

Here `inj`$_1$ and `inj`$_2$ are the two constructors allowing to build an element of a sum of types (either from an element of the first or second type). This leads to the commutativity proof:

```
comUnion : ∀ {c} → Symmetric (c ≡_∪_)
comUnion (prop₁ , prop₂) = (λ {(x , Tx) → prop₁ (x , flipSum Tx)}) ,
  (λ y → case prop₂ y of λ {((x , Tx) , x≈'y) → (x , flipSum Tx) , x≈'y})
```

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

```
subUnionₗ : ∀ {c c₁ c₂} → c ≡ c₁ ∪ c₂ → c₁ ⊑ c
subUnionₗ (prop₁ , _) (x , Tc₁x) = prop₁ (x , inj₁ Tc₁x)

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

Unicity of union We can prove the union is unique relatively to the clock equality defined earlier. 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 (_     , _) (_ , prop₄) x with prop₄ x
uu (prop₁ , _) (_ , _    ) _ | y , _ with prop₁ y
uu (_     , _) (_ , _    ) _ | _ , x≈y | z , y≈z = z , trans≈ (sym≈ x≈y) y≈z
```

We conclude by applying the previous property both ways.

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

Commutativity of intersection The intersection is also commutative:

```
comInter : ∀ {c} → Symmetric (c ≡_∩_)
comInter (prop₁ , prop₂) =
  (λ x → case prop₁ x of λ {(y , z , x≈y , y≈z) → z , y , y≈z , x≈y}) ,
  (λ y z x → case (prop₂ z y) (sym≈ x) of λ {(t , t≈z) → t , trans≈ t≈z (sym≈ x)})
```

Intersection and subclocking If c is enforced to be the intersection of c_1 and c_2, then c is a subclock of both of them, which can be proven.

```
subInterₗ : ∀ {c c₁ c₂} → c ≡ c₁ ∩ c₂ → c ⊑ c₁
subInterₗ (prop₁ , _) x with prop₁ x
subInterₗ (_     , _) _ | y , _ , x≈y , _ = y , x≈y

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

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 (_ , _    ) (prop₃ , _) x with prop₃ x
ui (_ , prop₂) (_     , _) _ | y , z , x≈y , x≈z with prop₂ y z (trans≈ (sym≈ x≈y) x≈z)
ui (_ , _    ) (_     , _) _ | _ , _ , x≈y , _   | t , t≈y = t , trans≈ x≈y (sym≈ t≈y)

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

Intersection and union As a consequence, we can 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≡c₁∩c₂ c≡c₁∪c₂ = trans⊑' (subInterₗ c₀≡c₁∩c₂) (subUnionₗ c≡c₁∪c₂)
```

7 Related work

We provide a mechanization of the semantics of CCSL in a proof assistant. As such, this approach could be reused for other concurrent languages. Such a work has already been done using different kind of formal methods, for example [7] using Higher Order Logic in Isabelle/HOL; [6] and [13] using the Calculus of Inductive Constructions in Coq, whose description can be found in [2]. The use of Agda in this development is motivated by the expressiveness of the language 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 [12], [8] and [3]. Although they differ from these two aspects, both of these tools rely on the same underlying intuitionist type theory, first described in [9] and clarified in [10].

The denotational semantics of CCSL on which this work is based can be found in [4]. TimeSquare, the tool developed to describe CCSL systems as well as solve constraint sets has been presented in [5]. As for CCSL itself, it was first presented in [1]. Although 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 [15] - which was translated from a constructive mathematical set theory to a generic type to better match the use of a type theory. Other attempts at giving semantics to languages like CCSL have been developed, such as a promising approach to give an operational semantics to TESL that can be found in [14].

8 Conclusion

8.1 Summary

In this work, we have proposed a mechanization of CCSL in Agda. We have clarified some notions inherent to this language (and even detected and corrected an issues in the paper version of the denotational semantics), and have proposed ways of encoding it in a proof assistant. Details about the lifetime of a clock, encoded as a birth instant and a death instant have been omitted. Their presentation would not have been suitable to this article. However, they have been encoded in the framework and will be presented in another paper. This work stands as an example of mechanization in Agda for a concurrent language, as well as an attempt to provide the CCSL developers with a complete mechanized semantics from which different features could eventually be extracted, as explained in the next section.

We advocate that mechanizing such semantics is mandatory when studying complex languages and systems, as standard paper semantics suffer from a lack of precise and complete formal and assisted verification.

8.2 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 are currently extending the language through the definition of instant refinement [11] in order to ultimately 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 the loss of some algebraic properties which we would like to handle properly.
– We will handle relations and expressions specific to discrete clocks. This requires to properly model these clocks which can be defined on infinite sets of instants while necessarily having a finite set of ticks. This is currently being investigated through the use of extensional equalities.

Acknowledgement

The authors would like to thank the CCSL team at INRIA for providing them with their time and valuable expertise regarding this language.

References

1. André, C., Mallet, F.: Clock Constraints in UML/MARTE CCSL. Research Report RR-6540, INRIA (2008)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series (2004)
3. Bove, A., Dybjer, P.: 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. pp. 57–99 (2008)
4. Deantoni, J., André, C., Gascon, R.: CCSL denotational semantics. Research Report RR-8628 (2014)
5. Deantoni, J., Mallet, F.: TimeSquare: Treat your Models with Logical Time. In: TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012 (2012)
6. Garnacho, M., Bodeveix, J., Filali-Amine, M.: 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)
7. Hale, R., Cardell-Oliver, R., Herbert, J.: An embedding of timed transition systems in HOL. Formal Methods in System Design **3**(1/2) (1993)
8. Malakhovski, J.: Brutal [meta]introduction to dependent types in agda
9. Martin-Löf, P.: Intuitionistic type theory.
10. Martin-Löf, P.: Intuitionistic type theory. notes by giovanni sambin.
11. Montin, M., Pantel, M.: Ordering strict partial orders to model behavioural refinement. In: Proceedings of 18th Refinement Workshop 2018, affiliated with FM 2018 and part of FLoC 2018 (2018)
12. Norell, U.: 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)
13. Paulin-Mohring, C.: 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)
14. Van, H.N., Balabonski, T., Boulanger, F., Keller, C., Valiron, B., Wolff, B.: A symbolic operational semantics for TESL - with an application to heterogeneous system testing. In: Abate, A., Geeraerts, G. (eds.) Formal Modeling and Analysis of Timed Systems - 15th International Conference, FORMATS 2017, Berlin, Germany, September 5-7, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10419, pp. 318–334. Springer (2017). https://doi.org/10.1007/978-3-319-65765-3_18, https://doi.org/10.1007/978-3-319-65765-3_18
15. Winskel, G.: 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)