Universiteit Leiden
Opleiding Informatica

The Constraint-Relation Modelling Language

and its relation to Petri Nets

Name: T. Buitenhuis
Studentnr: 0143235

Date: 06/03/2014

1st supervisor: Dr. Ir. F.J. Verbeek
2nd supervisor: Dr. H.C.M. Kleijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

The Constraint-Relation Modelling Language and its relation to Petri Nets

Bachelor Thesis

T. Buitenhuis

06/03/2014

Abstract CRML is a new graphical Modelling language for describing changing systems. This is a description of CRML, with explanations of design decisions that were made, and an investigation of the relation between CRML diagrams and Petri nets. The comparison to Petri nets helped define the details of the syntax and semantics of CRML.

Keywords: Constraints, CRML, Modelling languages, Petri nets

Contents

List of Figures

Chapter 1

The Constraint-Relation Modelling Language

1.1 Introduction

In this paper we will explain and discuss the syntax and semantics of the graphical modelling language CRML, and compare CRML to Petri nets. Furthermore, we will discuss some examples of CRML models.

1.1.1 The History of CRML

CRML was at first developed as an informal ad hoc language to describe constraints that copyright licenses put on movement of data, after a combination of data flow diagrams, UML use case diagrams, and UML class diagrams was found too limited for that purpose.

The first version of CRML had a positional notation of object attributes that assumed objects resemble contracts, this was replaced by elements similar to those in use case diagrams to make the language suitable for modelling the permissions subsystem of a cooperative writing and translation system[Bui08].

Some concepts related to time and numbers of objects were added after that.

Finally, the syntax and semantics of CRML were defined more precisely during the writing of this paper.

The current version of CRML appears to be useful for modelling distributed software, organisations and contracts, behaviour of cells, and possibly many other things.

1.1.2 Why (not) CRML?

The language was designed to be suitable for expressing *changing* structure, *changing* behaviour, relations between structure and behaviour, metarelations[1] and concurrency.

Although its name contains the word 'constraint', CRML is *not* a constraint programming language. A program written in a constraint programming language tries to achieve a goal within certain constraints. CRML models do not define such goals in

[1] For example, if Alice demands that Bob demands something from Chris, then there is a relation between Alice and the relation between Bob and Chris. Such metarelations are found in many copyright licenses.

any way, but may still be useful for modelling the constraints. CRML is more similar imperative and functional programming languages with guards[2].

If the problem being modelled involves only changing structure, but not the other listed purposes, or only changing behaviour, but not the other things, then one should consider using a different modelling language. Furthermore, no model checking, simulation, or code generation tools for CRML exist, and a complete method to translate CRML to the language of an existing tool does not exist yet either. CRML is primarily meant for documentation and communication.

1.1.3 Requirements

In the early sketches of copyright licenses, which were not meant for publication, CRML was jokingly called *Confusing Rights Modelling Language*[3]. To make CRML useful as a general purpose modelling language, and suitable for use by others, its ad hoc definitions had to be replaced by definitions that followed from certain design principles. Much like a computer program, a modelling language should not only be based on correct logic, but should also have a *user interface* suitable for its purpose. As mentioned in section 1.1.2, the purposes of CRML are to aid understanding of a certain type of problem, and to support communication.

- For both purposes a language is required to be readable: more information should be understood at a glance than is possible from written text ("a picture is worth a thousand words").
- The language also is required to be easy to put in a graphical presentation.
- It should be possible to make changes to diagrams without erasing and redrawing too much.

Although the possibility of creating visual programming and model checking tools is not required, it should not be made impossible, because an ambiguous modelling language is less useful than an unambiguous one.

- It is not required to make it easy to implement an automated method to translate CRML diagrams into software, a test framework, or a runable or queriable model.
- It is however required to make it easy for a programmer to write a program that implements the rules described in a diagram, therefore ambiguity is undesirable.

Apart from being readable, writable, editable and unambiguous a modelling language is required to be easy to learn.

- Use of familiar symbols from modelling languages that are likely to be known by the user makes learning a new language easier.
- The meanings of familiar symbols are not required to be exactly the same as in other languages, they can also be different but using familiar metaphors[4].

[2] *Guard-Relation Modelling Language* might have been a more accurate name, on the other hand guards and constraints are such similar concepts that guards are called constraints in *Mathematica*[Wol].

[3] The name *Constraint Relation Modelling Language*, although fitting, was mainly chosen to keep the acronym.

[4] For example arrows usually represent movement or point ('aim') to something, and rectangles ('boxes') often are containers.

- It is preferred to avoid introducing false friends, which are symbols or words that look the same but have different meanings in different languages. Unfortunately false friends can be impossible to avoid when they already exist as such in other languages. For example the meanings of rectangles and ellipses in UML use case diagrams are not the same, if not opposite, to the meanings of rectangles and circles in Petri nets.

CRML borrows a lot from UML[Obj], including the somewhat counter-intuitive[5] use of arrows for inheritance relations (see section 1.2.3).

1.2 Drawing CRML Step by Step

This section will introduce the basic elements of CRML diagrams, in the order they would be drawn in a new diagram (first the elements between which relations exist, then the relations). It also explains the rationale of the elements.

1.2.1 Object Descriptions

CRML is meant to model both behaviour and structure. It is therefore by nature an object oriented language, and the main type of element is the object description. (see figure 1.1).

Unlike in many other object oriented languages, an object description describes not what *is*, but what *may be*. A group of actual objects to which a CRML diagram pertains is called a world. By comparing an actual object to the object descriptions in a diagram, one can draw conclusions about what behaviour is possible for the actual object. After performing that behaviour, the object may match a different object description than before. Thus, CRML diagrams describe rules, not facts. This way it is not necessary to draw the resulting situation from each and every possible behaviour. Since it is necessary[6] to allow both object structure and behaviour to change *and* to allow both to influence the other, the number of possible behaviours could potentially be very large, which makes it unfeasible to draw every possible state.

The Header

Object descriptions have a class (this could for example be 'Person', 'Company' or 'License'), and are members of a set. The set of all objects that belong to the same class is called U (from Universe), or U(*SomeClass*) to distinguish the set of all objects belonging to U(*SubClass*) from its superset U(*SuperClass*), where *SubClass* inherits[7] from *SuperClass*. When there is no set explicitly written in the header, it is implied to be U. In place of a set, one may write an expression that can be evaluated to a set.

[5] The direction of the arrow is left to right in the sentence '*ChildClass* is a kind of *ParentClass*', but when one thinks of inheritance as the copying of an interface, then the arrow is in the opposite direction of the copy. Furthermore, parent classes tend to be implemented before child classes, in that sense the arrow is pointing backwards in time.

[6] For example one may want to describe acquiring a work with a copyright license attached to it (behaviour changing structure), and that license containing restrictions on what one may do with other data one links to the work (structure changing permitted behaviour).

[7] Inheritance is described in section 1.2.3.

Examples: $U \setminus \{'Fred'\}$ means everyone except Fred[8], and $\{queue(1)\}$ could be the set containing only the first person in a queue. The function *queue* is not part of CRML and would need to be defined outside the diagram.

A set may be a free variable. After comparing an actual object to the diagram and binding sets to the free variables to make it match, one should forget the bindings before evaluating a different actual object.

Example: One may declare that two object descriptions cannot refer to the same actual object[9] by making the set of the one X, and the set of the other X^c (as in figure 1.2). After deciding that a certain actual object is in X when considering its behaviour in relation to another actual object that is decided to be in X^c, it does not remain in X, and it may simultaneously be in X^c from the perspective of the other object if that believes itself to be in X. For example a diagram could forbid self reference, e.g. "every person can only give handshakes to people who are not the same person". From the perspective of each person, the set 'everyone else' is different. Since the statement applies to everyone (U), but it is not possible to use U to refer to the person shaking hands because he cannot give himself a handshake, it is clearly needed to allow free variables in the language if one wants to be able to express such statements.

In the first version[10] of CRML that used elements similar to those of UML use case diagrams (we will discuss those elements in section 1.2.1), free variables referred to single instances of objects and not to sets. This made it necessary to have an identity field in addition to the class and set fields, for which we found no purpose that could not also be served by allowing the set to be a free variable.

Examples: *AClass x U* in the old notation has a lowercase identity field x, which is a free variable, whereas the set U is not a free variable. The expression is equivalent to *AClass X* in the newer notation.
AClass x $\{x\}$ did not mean that there could be only one object at a time that fit the description the header was part of, that combination of identity and set would have been meaningless because an object always is a member a set that is defined as containing that object. In the new notation, however, the $\{X\}$ in *AClass* $\{X\}$ does refer to (a set containing a) single object, and that X is a free variable itself which may match multiple objects – relations will in this case be to one object, but multiple instances of the situation that object is in may exist concurrently. A way to express that only one object may be in a certain state at a time will be introduced in section 1.2.4, although only in relation to another object (which may of course be a simple one named something like "everything").

If an item in a set is written between quotes (for example as in $\{'Fred'\}$), that item is not a free variable but refers to a specific object.

One may be tempted to use {} (the empty set) to express that *no* object may fit a certain description. The semantics of CRML do not work that way – all that {} means is that one might just as well not have drawn the object description in the header of which one placed the {}.

[10] See section 1.1.1 for a short description of the order in which changes were applied to CRML.

The Inventory and Activities

Apart from a class and a set an object description optionally contains an inventory, which is a description of the structure of the object, and activities, which are behaviours the described objects may perform.

Activities are represented by ellipses. If a behaviour occurs at regular intervals[11], its frequency may be added to the name of an activity, for example *lunch*(24*h*).

Activities are atomic: they do not have a start and end time but are considered to occur within a single moment. There are a few reasons for this:

- If activities were not atomic, they would be equivalent to object descriptions (an object that exists is in a sense performing the action "existing") and thus redundant.
- Activities have a start and end time in the sense that they can be enabled and disabled. Them being atomic avoids confusion between beginning or stopping to be able to do something and beginning or stopping to do it.
- Activities being atomic simplifies reasoning about concurrency, and allows the user to not define the amount of time an activity takes, which reduces the complexity of typical models.

When it is necessary to model a start and end time, one can let an activity disable itself and enable another one (using activity-state relations, which will be introduced in section 1.2.2). This second activity then both re-enables the first activity and disables itself with a frequency equal to the time between start and end: the first and only time it is performed (before being enabled again) marks the end time.

Both sections[12] may be left out, and it is not necessary to label them because their contents look different, however it is recommended to always label the sections to avoid any confusion. If possible without making the diagram difficult to read, it is recommended to draw the inventory on the left, doing so puts the beginning of data flow (see section 1.2.2) within a single object description in left-to-right order. If the user, in order to draw faster, prefers not to label the sections, it is recommended to still do so when both exist and the inventory is not on the left.

An object description without activities is called a passive object description, and an object description that does include activities is called an actor description.

The inventory of an object description contains other object descriptions.

If actor descriptions are inside an inventory, then they are said to be acting on behalf of the object that contains them.

> **Example:** In figure 1.3, a company has employees, and when it sends a package to a customer, it is not the company itself that sends it, but one of its employees.

[11] Requirement relations, which will be introduced in section 1.2.2, may prevent an activity from being performed. In such cases, it is skipped.

[12] The word *section* was chosen when CRML was meant to model documents. We could choose different jargon; but in software a *component* is something bigger than an object, and in general a *part* is not necessarily a container. The word *element* has the same problem as *part* and in addition to that an object description is one type of CRML element, whereas inventories and activity sections are not diagram elements but parts of one type of those.

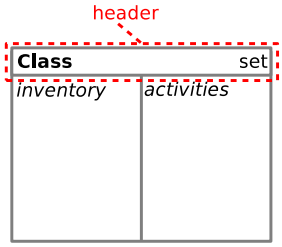

Figure 1.1: An object description

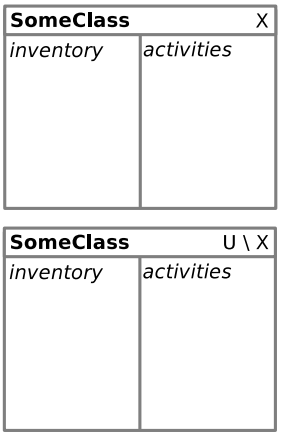

Figure 1.2: Two actor descriptions that cannot both refer to the same instance

If an object description overlaps the inventory of two (or more) other object descriptions, it is part of the inventory of both, and is a connection between the two. An object description may also be partially inside another, and partially be a top level object description. For an example of such shared inventory, see appendix A.2.

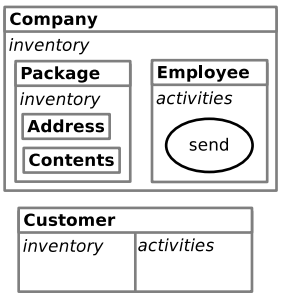

Figure 1.3: A hierarchy of inventories

1.2.2 Activity-State Relations

With the concepts described so far one can describe static structure and, to some extent, behaviour. To describe *changing* structure and behaviour, one needs to show both how structure affects which activities can be performed, and how activities change structure. There are two basic types of relations that describe these, each with two subtypes that are each others complements. Furthermore there are relations that are combinations of these types. The complete list of activity-state relation types is:

- state modifying relations
  - provides
  - revokes
- requirements
  - requires
  - forbids

  Requirements may have identical arrowheads on both sides when the same relation exists in both directions.
- combined, binary
  - consumes
  - blocks
- combined, ternary
  - copies
  - blocking-copies
  - moves

 blocking-moves

We will see why there are no[13] other combined relations in section 1.2.2.

State Modifying Relations

State may be modified in three ways:

- Inventory is added or removed.
- The ability to perform an activity is gained or lost.
- A relation is created or destroyed.

Both in case of *provides* () and *revokes* (), the arrowhead points at what is provided or revoked, and the other end of the relation is attached to the activity that is modifying the state of the world. It is allowed to provide something that already exists, or to revoke something that doesn't exist. Those two actions have no effect on the state of the world.

When there is no *provides* points at an inventory item, activity or relation, that means whatever it represents must exist in the initial state of the world. However, if there is a *provides* pointing at a diagram element, then what it represents does not exist in the initial state, unless there is a copy of the diagram element that isn't pointed at by a *provides*.

The result of simultaneously providing and revoking something is undefined. Diagrams in which this may happen have to be fixed with the *forbids* relation, which we will discuss in section 1.2.2.

Requirement Relations

Most real-world activities are only possible when certain requirements are satisfied. In CRML, this is expressed by *requires* () and *forbids* (). The arrowhead points at what is required or forbidden for an activity to be possible, and the other end of the relation is attached to the activity. An activity can only be performed when all its requirements are satisfied (and everything that it forbids is not true), and the state modifications performed by an activity do not affect the satisfaction of the requirements (for example an activity revoking something that it requires prevents it from being performed again, but not from being performed at all).

If either one requirement needs to be satisfied, or another, but not necessarily both, then a requirement can be forked as in figure 1.4.

Forks with a square are logical *or*s: at least one sub-requirement must be satisfied to satisfy such a forked requirement.

Forks with a circle are logical *and*s: all sub-requirements must be satisfied to satisfy this type. *And*-forks are not strictly necessary, but converting a list of requirements to *conjunctive normal form* (single requirements and *or*-only expressions linked by *and*s) could lead to large and unreadable diagrams.

A *requires* that points at an activity means that activity must be performed concurrently with the one it is a requirement of, and a similar *forbids* means they may never be performed concurrently.

[13] In section 1.2.4 we will see more are possible after introducing a language feature not discussed yet.

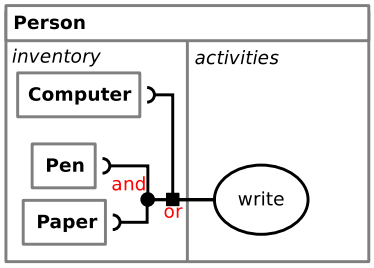

Figure 1.4: Alternative requirements

A *forbids* with only one arrowhead between two activities gives priority to the one on the side of the arrowhead, whereas when two activities forbid each other, neither has priority.

There is a symmetry in the rules for *requires* and *forbids*: in the case of *requires*, the activity on the side of the arrowhead is "more important" in the sense that it can be performed without the other, and in the case of *forbids*, the activity on that side is "more important" in the sense that it is given priority.

Requirements Relations, State Modifying Relations, and Concurrency

Although intuitively requirements are like preconditions and state modifying relations are like postconditions, the requiring/forbidding actually is not performed before the providing/revoking but simultaneously. *Blocks* and *consumes* are possible because requirements ignore simultaneous state modifications by combined relations they are part of (as well as by other state modifying relations belonging to the same activity), not because one happens before the other (remember that activities are atomic!).

Inherent Attributes

The side of an activity-state relation without an arrowhead may also be attached to an object description or a relation instead of an activity. The 'activity' in that case is the act of existing. Such relations are called inherent attributes. The inherent attributes affect structure and behaviour according to the following rules:

- When an inherent attribute provides something, it will exist until the providing element is revoked, unless the provided diagram element is also provided by another inherent attribute before that happens. Activities may revoke the provided object, but this has no effect.
- Similarly, when an inherent attribute revokes something, the revoked element cannot exist until the revoking object is revoked itself, even when the revoked element is provided by an activity.
- When a diagram element is both inherently provided and inherently revoked, like in case of the regular *provides* and *revokes* relations, its existence is undefined and the diagram has to be fixed using a requirement.
- When an inherent attribute is a *requires* or *forbids* relation, the requiring element will only exist when the condition is satisfied. A good use for this is a

requirement attached (with the side without an arrowhead) to a state modifying relation. In such cases, it is possible to perform the activity the state modifying relation is attached to when the requirement isn't satisfied, but the state modifying relation will only have an effect when the requirement is satisfied. Similarly, requirements may have such metarequirements.

An inherent attribute cannot require an activity. The requiring object would only exist while the activity was performed, but since activities are atomic (see section 1.2.1) the time between their beginning and end doesn't exist in a CRML model.

It is not necessary for an object description to explicitly require another object description that contains it, not even when it's nested multiple levels deep. Rules only apply in the type of location they are drawn in, except for top level rules, which apply in any location[14] as long as no relation has to cross an extra object description border. If one wants to require a containing object description, one probably actually would like to require another object that is also inside the containing object.

> **Example:** Instead of saying a teacher requires a classroom to teach, one should say a teacher requires a blackboard to do that. It is possible to teach in a different kind of location if there is a blackboard, and it is not possible[15] to teach in a classroom where there is no blackboard.

Combined Relations

There are two kinds of combined relations:

- The binary combined relations *consumes* ( ) and *blocks* ( ) are equivalent to a requirement and a state modifying relation existing between two diagram elements.

  Consumes and *blocks* are the only binary combined relations, it is self-contradictory to include both state modifying arrowheads or both requirement arrowheads in a combination. The remaining two combinations of arrowheads are revoking something that must not exist, and providing something that must already exist, which are meaningless.

- The ternary combined relations (*copies* ( ), *blocking-copies* ( ), *moves* ( ) and *blocking-moves* ( )) have more meaning than the relations one would split them into: A curve 'bouncing against' an activity like in figure 1.5 means both ends of the relation pertain to the same object, or an exact copy of it, not just to an object of the same class.

  Ternary relations move objects, so one end must be *provides* or *blocks*. It's not possible to move something that does not exist, so the other end must include *requires*.

The following rules apply to combined relations:

- An activity can only be in the middle (bounce) of a ternary relation, it is not possible to move or copy an activity.

[14] In the real world, there is no top level, any object is inside a bigger one. Only the astronomical universe could be considered to be at the top level, but it cannot interact with anything outside it.

- Combined relations cannot be inherent attributes. The binary combined relations would be self-contradictory as inherent attributes, and the ternary combined relations would make no sense: in CRML an object cannot be partially in one place and partially in another[16], therefore movement and copying have to be atomic.
- Combined relations cannot point at activities: it would be meaningless to demand an activity is not already being performed when it is made possible, it is not possible to revoke an activity while it is being performed (but it can be revoked afterwards), and behaviour isn't structure so activities cannot be moved.

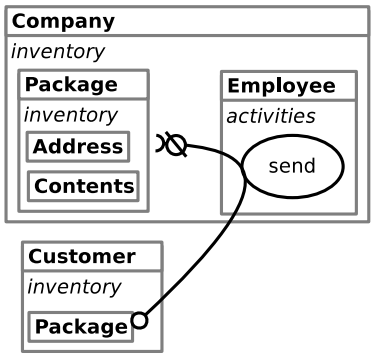

Figure 1.5: An example of a ternary relation

1.2.3 Other Relations

There are two more types of relations: the link (.....), and inheritance (⟶▷).

Links

A link is drawn between two object descriptions when they are connected in some way without one being part of the other. It is recommended to use links only when it's not possible to put both object descriptions in the inventory of a third object description. One may consider using shared inventory (see section 1.2.1) instead of a link.

Links can be provided, revoked, required and forbidden.

Inheritance

An inheritance relation can exist between two activities or between two object descriptions, to show the inheriting class or activity is a 'kind of' the inherited. It might often not be important to show that relation, but then it is still useful to save space.

[16] It can, however, be in multiple places at once as shared inventory.

The following rules apply to inheritance:

- Inheritance cannot[17] be provided, revoked, required or forbidden.
- An activity that inherits from another receives all the same relations that are connected to the one the arrowhead points at. For example 'fire' could inherit from 'demote'.
- An object description that inherits from another receives all inventory, activities and relations that are part of the one the arrowhead points at, and its class will be considered a subclass of the class of the object description it inherits from. The universe is always implied to be that of the class of the object description itself, not that of a superclass. In some cases one may need to write something like $U(Super)$ (in which *Super* is the name of the superclass of the class of the object description).

1.2.4 Slots and Stacks

So far the possibility that multiple objects of the same class may have different roles that cannot be modelled using subclasses (which would be defined by inheritance) has not been discussed.

> **Example:** An example of roles not fitting the use of subclasses exists in the famous dining philosophers problem (which is described but not attempted to solve here). Each philosopher needs a fork in each hand to be able to eat, is unable to move a fork from one hand to the other, and has to share the fork in his left hand with the right hand of the philosopher to his left. Using subclasses one could model left hand forks and right hand forks, but in the dining philosophers problem each fork is both.
> To make clear which of the two forks is meant, the forks in the inventory of a philosopher need to be labelled.

A labelled object description is called a slot. The following rules apply to slots:

- Any relation that is possible for a normal object description is possible for a slot. In addition to that it is possible to move an object from one slot to another within the same inventory.
- There can be only one instance per location of a labelled object. Labelled object descriptions obviously do not need and cannot have a set.
- If there is a slot of a certain class in an inventory, then all object descriptions with that class in that inventory (excluding those nested inside other object descriptions) must be slots.

  Otherwise it would be unclear what should happen to an object in the slot when the non-slot object description with the same class is provided or revoked. We could define rules for these situations, but forbidding them keeps the language simpler.

[17] Metarelations to inheritance will be allowed in a future version of CRML. Currently especially requirements of inheritance are considered too confusing, because although every class is a 'kind of' itself, that is not shown in diagrams, which will lead to that fact being overlooked. This problem will probably be resolved by distinguishing between interfaces and instantiable classes, by marking the latter with the same symbol that is the inheritance arrowhead.

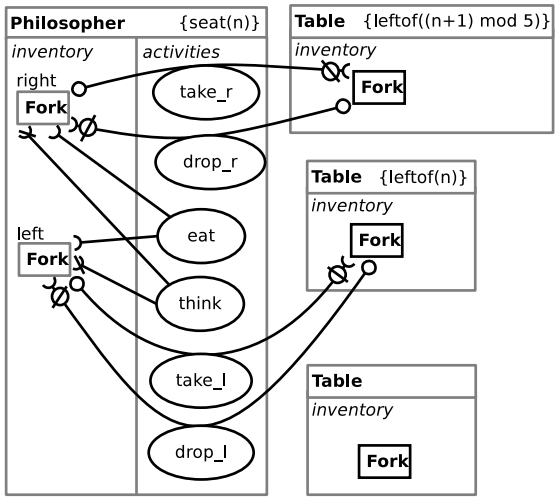

Figure 1.6: The dining philosophers problem

Apart from their role, the number of objects of a certain class (or indeed the number of objects in a slot) may be important. For this purpose, CRML has stacks.

Stacks may be slots, such stacks are called named stacks. The number of items on an unnamed stack includes all unnamed stacks of the same class that are in the same inventory. To set the initial height of a stack, one must name it and write the height behind the name, for example *cash*(50).

Objects on a stack lose part of their identity:

- Links cannot be attached to stacks.
- Stacks cannot share inventory.
- Stacks cannot have a set.

Multiplicity of Relations

Activity-state relations connected to stacks must have multiplicities, which are the numbers of actual objects affected by each side of the relations. The term multiplicity is borrowed from UML[Obj] (it probably also exists elsewhere), a similar term used in other modelling languages is 'weight'.

The following rules apply to multiplicities:

- A multiplicity may be a number, a variable or *all*. *all* means all items on a stack will be revoked, moved or copied.
- When there is no multiplicity annotation, the multiplicity is implied to be one, except for the multiplicity of an inherent *revokes* pointing at a stack, which is always *all*.
- Stacks can have inherent attributes.
  - The multiplicities of those are multiplied by the number of items on the stack.

– If an inherent attribute of a stack is a diagram element that isn't a stack, it exists when the stack isn't empty.

The meaning of multiplicities is defined as follows:

- For *provides* and *revokes* the multiplicity is the number of objects provided or revoked.
- For *forbids* it is the maximum number of objects that is allowed, plus one[18].
- For *requires* it is the minimum number of objects required.
- If a state modifying relation to a stack is an inherent *provides*, then while the providing object exists, the number of objects is at least the provided number (revoking more will have no effect).
- If a requirement to a stack is an inherent attribute, the requiring object will cease to exist when the requirement is not satisfied.

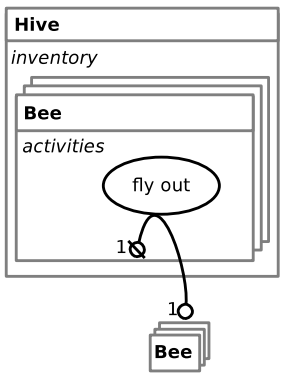

Figure 1.7: A very simple example of a stack

Multiplicities of Forked Requirements Forked requirements can have multiplicities.To accomplish this, write variables next to the arrowheads and a sum expression next to the fork, for example $c = a + b$ or $a + 2 \times b = 4$, where a and b are the variables written next to the arrowheads. For an example see appendix A.2.

Multiplicities of Combined Relations Again it is possible to make combined relations, we will not list all of them (it is now possible to have both a *requires* and a *forbids* on the same side of a relation, for some multiplicity combinations *forbids-and-revokes* and *requires-and-provides* will have meaning. . .). Some combinations may be inherent attributes.

[18] If for example the multiplicity is 5, then it is forbidden to have 5 objects on the stack, but allowed to have 4 or less

The following rules apply to the multiplicities of combined relations:

- If a single multiplicity is written at a combined end, all relations expressed by arrowheads at that end have that multiplicity. In the unusual case of different multiplicities for combined arrowheads, these should be listed in order from outermost to innermost arrowhead and be separated by semicolons[19] and no spaces.
- In case of a ternary combined relation, the multiplicity of all relations that are part of it is implied and required to be the same: a single group of objects cannot have multiple sizes.

[19] Quotients and floating point numbers may be used as multiplicities. This means slashes, colons, points and commas cannot be used as separators (some languages – Dutch is one of them – use the comma as a decimal separator).

Chapter 2

CRML Diagrams as Petri Nets

In this chapter it is assumed the reader already has basic knowledge of Petri nets.

Since no model checking or verification tools for CRML currently exist, it could be useful to translate CRML diagrams to Petri nets, for which mature software does exist.

However, the main reason to attempt to translate CRML to Petri nets was to improve the semantics of CRML, a step towards a formal definition. While that definition does not yet exist, logical verification of the translation method is not possible, a chicken and egg problem. Furthermore, we will see that an automated translation method that can be applied to any CRML diagram is improbable.

2.1 Comparison

2.1.1 Element Shape Confusion

Petri nets were already mentioned in section 1.1.3, where the opposite meanings of boxes and ellipses (or circles) in UML use case diagrams and Petri nets were used as an example of *false friends*. There are two reasons CRML uses boxes and ellipses the way UML does instead of the Petri net way:

Firstly, Petri nets were not considered while designing CRML, although having knowledge of Petri nets may have influenced the design.

Secondly, drawing diagram elements within other diagram elements is easier when the outer elements are rectangles. This is because we are used to drawing on rectangular surfaces (sheets of paper, blackboards, windows in drawing software...), and also because we are used to aligning objects in rows, columns and grids, which is a good way to save space inside a rectangle but not inside an ellipse.

2.1.2 Worlds and Markings

A Description of All Petri Nets

A CRML diagram represents a set of rules, about which certain properties may possibly be proved, but which cannot be simulated or executed in any way without a *world* to apply them to. In this way a CRML diagram is similar to an unmarked Petri net.

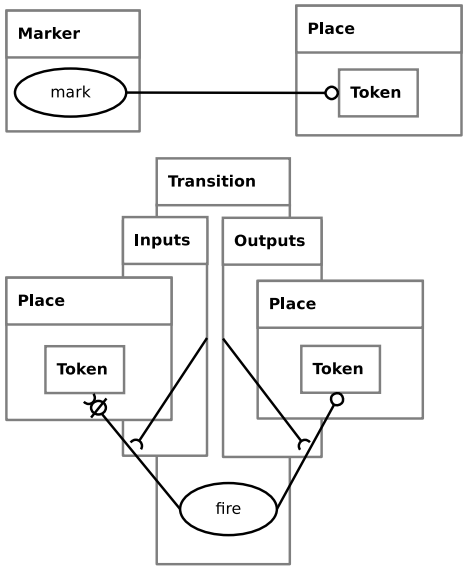

Figure 2.1: A CRML model of any Petri net

Figure 2.1 is a model that describes any traditional Petri net: tokens are created by an initial marking (this top part of the diagram may be left out to describe an unmarked net), and when a transition fires tokens are removed from its input places and created in its output places.

Note that in this model the firing of a transition destroys and recreates tokens, rather than transferring them. This is because the ternary relations of CRML cannot move or copy an object to multiple destinations at once.

The inherent requirement relations ensure that if a place is in the set of input places of a transition, a token will be consumed by the firing of the transition, and that if a place is in the set of output places, a token will be provided to it. In other words, the inherent requirements express that tokens must be consumed from and provided to *all* input and output places.

If there were an simple method to translate this model to a Petri net, there would exist a simple Petri net that explains most of what Petri nets *are*. Although this is not conclusive proof that such a method cannot exist, it is strong evidence[1] that not all CRML diagrams can be translated to Petri nets.

[1] The author of this report is not able to imagine what such a Petri net would look like.

Describing Specific Petri Nets

If one labels the places and transitions of a marked Petri net, creating an equivalent CRML diagram is trivial, as is illustrated in figure 2.2. If a CRML diagram can be modified to be similar to the ones in this example, the translation from CRML to Petri net will be trivial too. We observe that:

- In the first CRML diagram in figure 2.2 all object descriptions have sets without free variables and the initial stack heights are known: the world is completely defined. Such diagrams are called a explicit diagrams. Explicit diagrams may contain object descriptions without set annotations as long as those all have providing relations pointing them (which means those described objects don't exist in the initial state), or there is at least one object description of the same class that has a set description without free variables (that is, we can implicitly use the set U, which is not a free variable). Most CRML diagrams are not explicit, whereas Petri nets can be marked or unmarked, but not something in between.

  The second CRML diagram allows multiple instances of the Petri net (as well as zero instances). It is not completely explicit, but using this approach to translate from CRML to Petri net would require merging the entire diagram into one object description, which is likely to be impractical and gives no meaningful freedom to leave part of the world undefined.

- There are no functions defined outside the diagram (such as *queue*() in section 1.2.1), such expressions would require some creativity when translating from CRML to Petri net, automated translation is impossible because the definition of user-defined functions is not part of the definition of CRML.

- Only a very limited subset of all that exists and is allowed in CRML is used. Two limitations that stand out in this example are that objects inside inventories do not have inventories themselves, and that the only relations used are *consumes* and *provides*.

2.1.3 Missing Features

In section 2.1.2 we observed that CRML diagrams that are trivially equivalent to Petri nets don't use many of the features of CRML. To make translation from CRML feasible with the translated diagram still being recognisable in the result, we will have to use some extended type of Petri nets.

If analysis of the behaviour of the Petri net is to help us understand a typical CRML diagram that models changing structure, it needs to be possible to use nets as tokens in other nets, and to move token-nets into and out of other token-nets ([Val04] is a good introduction to such *object Petri nets*). It is not enough to have places or transitions that can be expanded to subnets, the structure has to change. Furthermore, the communication between nets has to happen through transitions: communication through places introduces a buffer, while a moved object disappears from one inventory synchronously with its appearance in another.

We will use the variant used in RENEW[KWDC09], it fits the above requirements and has arc types that are equivalent to common activity-state relations:

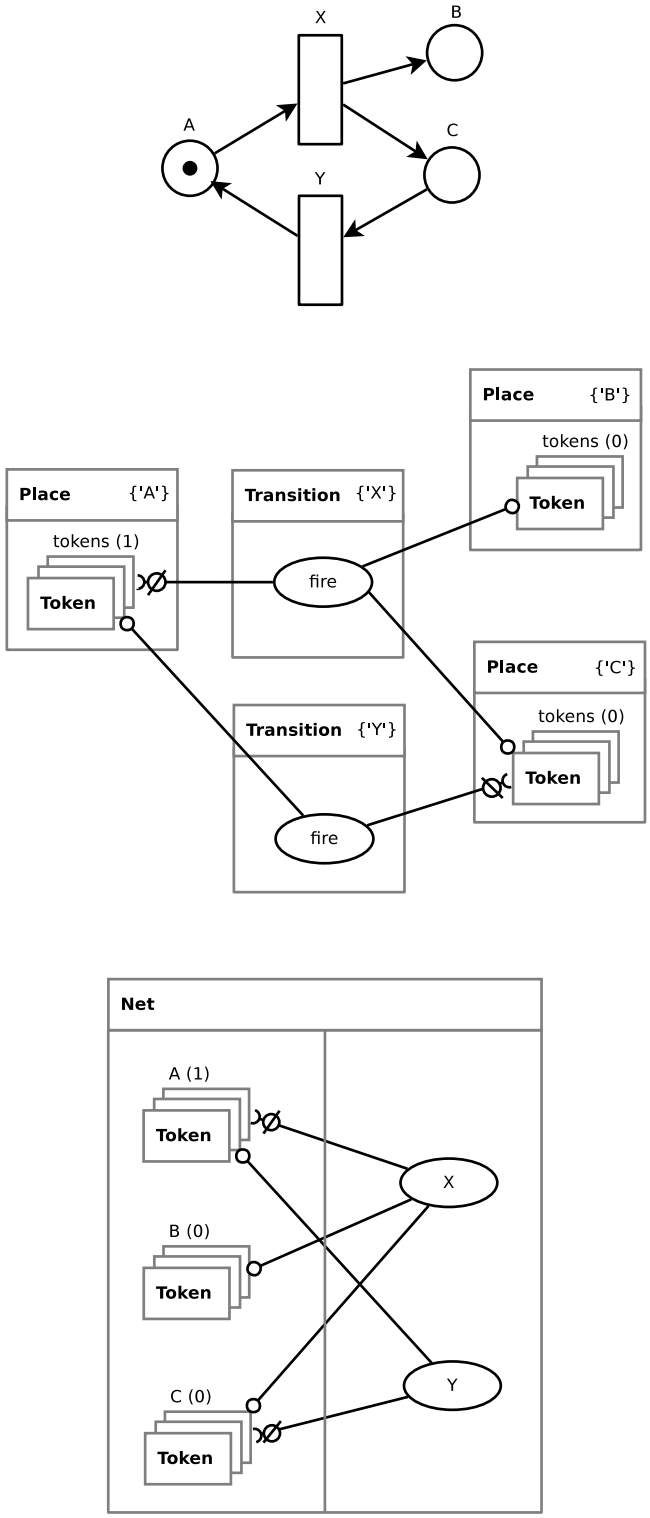

Figure 2.2: A Petri net and two equivalent CRML diagrams

RENEW		CRML
⟶	input arc	consumes 1
⟶	output arc	provides 1
⟷	reserve arc	does not exist
——	test arc	requires 1
⟶	flexible input arc	similar to consumes *n*, but does not exist
⟶	flexible output arc	similar to provides *n*, but does not exist
⟶	clear arc	revokes all
●—●	inhibitor arc	forbids 1

The reserve arc makes a token unavailable while a transition fires, which prevents other transitions that need the same token from firing synchronously with the one the reserve arc is attached to (unless the place contains at least one more token). CRML achieves something similar with a mutual *forbids* relation between activities. In addition to that, it allows expression of one mutually exclusive activity having priority over another by having the forbids-arrowhead only on the side that has priority. The reserve arc is equivalent to an input arc and an output arc. Combining the equivalents of those is not allowed in CRML. Since we want to translate from CRML to RENEW and not in the other direction, this is not a problem.

Flexible arcs specify which tokens are moved simultaneously, whereas relations with multiplicities only specify the number but not the identities or structure of the objects (although the structure of all moved objects may be matched to a single pattern rather than a list of complete structures).

We can create equivalents of more relations by combining arc types:

RENEW	CRML
n input arcs	consumes *n*
n output arcs	provides *n*
inhibitor arc and output arc	blocks 1
test or input arc, and clear arc	consumes all

Unfortunately we cannot construct *requires n* and *blocks n*, The *arc inscriptions* and *guard inscriptions* of RENEW (annotations of arcs that assign or match tokens to variables or tuples or lists of those, and annotations of transitions that must evaluate to true to allow the transition to fire, respectively) allow us to compare values but not to count them. *n* test arcs without inscriptions will be satisfied by a single token, with inscriptions we could only test for *n different* ones. Inscribed inhibitor arcs need other arcs (in this case output arcs) to assign values to their inscribed variables, the result is that they won't block the transition when the tokens to be added to the place are in any way different from the ones already there.

The equivalents of ternary relations can be constructed by inscribing the arcs through which tokens enter and leave a transition with the same variables (x, y):

RENEW	CRML
test arc x; transition; output arc x	copies 1
test arc x; transition; output arc x and inhibitor arc	blocking-copies 1
n input arcs $x, y \ldots$; transition; *n* output arcs $x, y \ldots$	moves *n*
input arc x; transition; output arc x and inhibitor arc	blocking-moves 1

There does not appear to be a straightforward way to translate the activity-state relations not listed the tables above to RENEW.

Timed activities (which are attempted at a regular intervals) aren't translatable either. Transitions may fire when their input requirements are satisfied, but there is no way to force them to try this a certain period after the last time it was attempted. RENEW does allow the user to specify at what time a token becomes available, but this can

only be used to trigger a finite number of repeats (because we cannot draw an infinite number of tokens), and the token will remain in its place when the time has passed. It is possible to construct a net that will remove a token if it hasn't been used (let a place with a token that appears slightly later be the other input of a transition with a clear arc to the place the first token is in), but that's still different from attempting an activity only at a precise time.

Shared inventory also cannot be translated. In RENEW it is possible to let a token exist in multiple places at once, but 'destroying' it will remove the reference (pointer) from one place but not the others. However, in many cases shared inventory is used to mark a group of diagram elements as an interface which is never revoked. In those cases it is possible to move the shared object description into one of the containing ones. Links are a special case of shared inventory, we can interpret them as empty object descriptions with class *Link*.

2.2 Translation from CRML to Petri Net

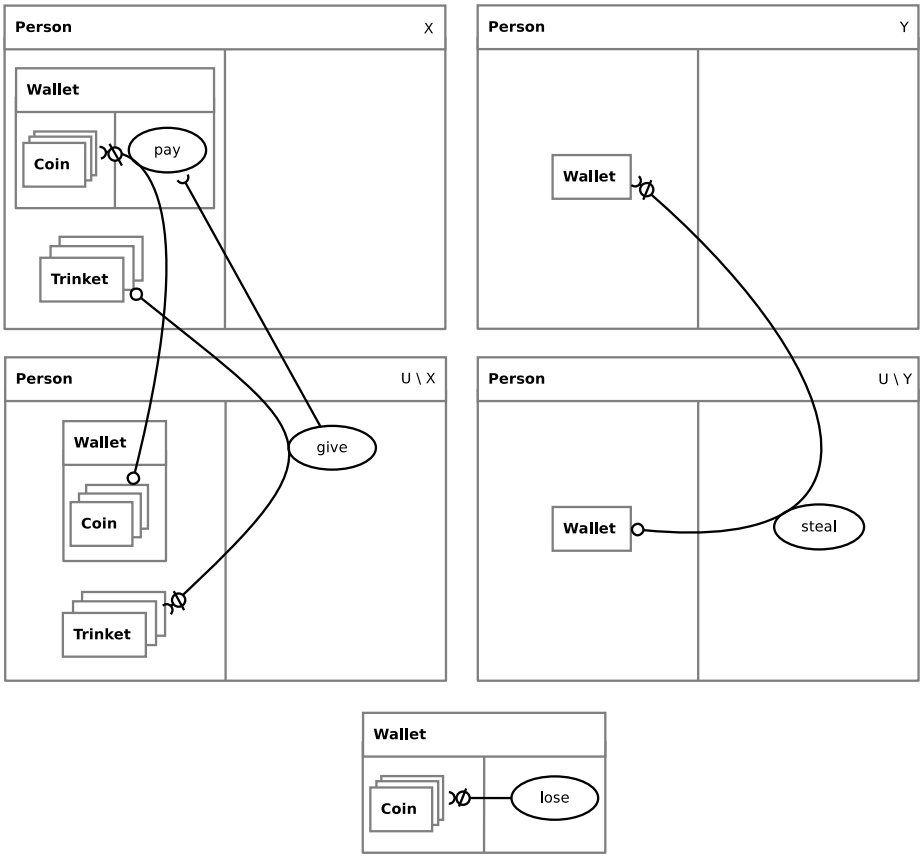

Figure 2.3: Trinket trade

A simple model (figure 2.3) of people buying objects (called trinkets to avoid confusion) from each other will be used as an example. Transfer of a trinket requires payment of one coin, and a wallet is needed to be able to pay. A person may steal another person's wallet. Coins may be lost at random.

If the diagram contains timed activities, mutually exclusive activities with priority, or any of the relations that cannot be translated according to section 2.1.3, we should give up now. If it contains links or shared inventory, we must attempt to fix that as described in section 2.1.3 before proceeding.

If the diagram contains object descriptions with copying relations attached to them, these cannot have an inventory or activities. RENEW uses references as tokens, and copying a reference will not create an independent instance of a net. It is possible to create another net with the same structure, but that will be in the initial state and not in the state the "copied" net was in. Copying references is equivalent to copying objects only when those objects cannot be changed in any way and cannot change themselves.

The difference between copying of references and of instances, as well as the other reasons to give up translation, can be worked around by modifying the diagram. Such

modifications will change the meaning of a diagram, and translation algorithms should not do that. A human can decide to change unimportant details of a model to make it suitable for describing in a certain language (or two languages), an algorithm cannot decide what is important.

2.2.1 Make the Diagram Explicit

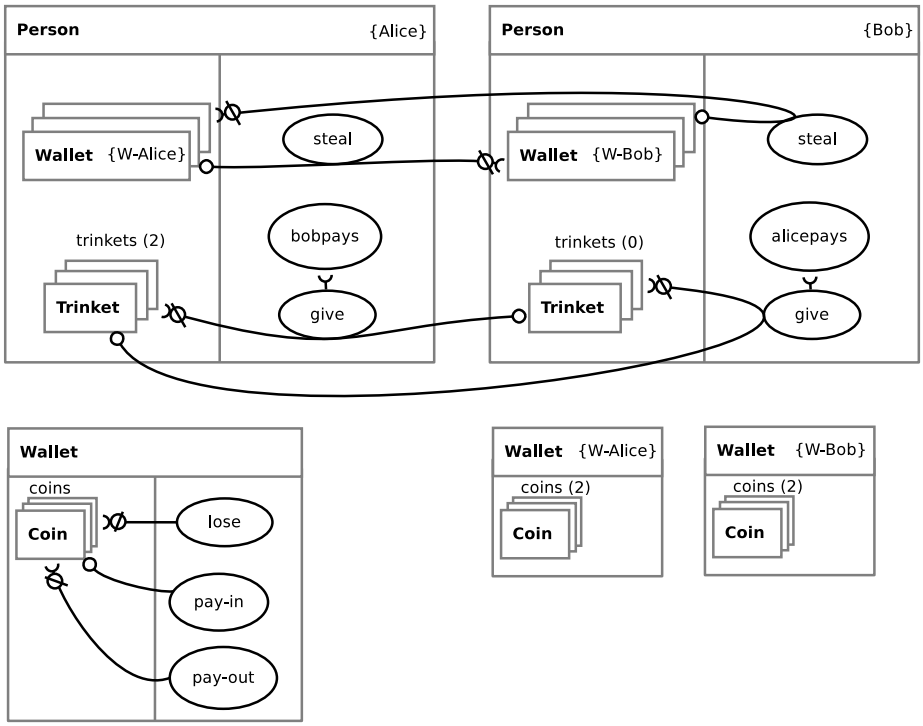

Figure 2.4: An explicit but not quite CRML diagram

The first step is to define a world and make the diagram explicit. It may seem attractive to make this the last step instead to keep the diagram as concise as possible for as long as possible, but being able to see what can and cannot change makes the process less confusing. Furthermore, we may be able to leave out parts of the diagram that involve classes of which we choose to have no instances.

In our example, there will be two instances of the class Person. Because any Person is either in the set X or in the set $U \setminus X$, and no *provides* points at Wallets in those, each Person will have a Wallet. We give each Person a number of Trinkets, and each Wallet a number of Coins.

We draw the diagram such that there is no nested structure[2].

If an activity involves nested object descriptions, it is split into multiple activities, each in one of the object descriptions its relations used to pass through. It is for now invisible that these activities must be performed synchronously and that objects that used

[2]Note that a top level object description with a known object identity does not contradict the same identity being drawn inside another object description's inventory

to be passed through ternary relations will preserve their identity. The only activities that have relations that cross borders of object descriptions are the ones that were at the top level in the original diagram, there is no containing object description that we could add an activity to.

Inheritance is eliminated by drawing all inherited elements there where they were inherited to[3].

Inventories may now only contain stacks. If there were any slots, *forbids 1* parts must be added[4] to the arrowheads of relations providing those – a stack that cannot grow higher than one item is equivalent to a slot. All providing relations that are possible to slots have blocking variants that have equivalents in RENEW.

Stacks may instead of numeric heights have set descriptions listing objects, which normally isn't allowed in CRML.

Our example now looks like figure 2.4.

2.2.2 Switch to RENEW Elements

We replace the CRML elements by their RENEW equivalents, and split the diagram into multiple nets: one for each class, or multiple for classes of which the objects may have different initial states. It is unclear whether RENEW allows a transition that creates a net-as-a-token to communicate with that net while it is being created in order to tell it its identity, which it would then use to initialise itself. Furthermore a net with an initialisation phase would be unnecessarily complicated because it would need to prevent unexpected behaviour before the initialisation is completed. It is simpler to draw a separate net for each possible initial state. To keep the example simple, we gave all the wallets in figure 2.4 the same number of coins.

The diagram now looks like figure 2.5. The variable names in the arc inscriptions are equivalent to the "bounce" in ternary relations, they show that the same reference is passed through the input and output arcs.

RENEW uses uplinks and downlinks to synchronise transitions. We use these both to move references between nets and as a translation of requirements between activities. A downlink consists of a reference (which we obtain using a test arc), a colon, a name, and a list of references that will be shared with an uplink. An uplink consists of the same starting from the colon, it will synchronise with any downlink that calls it.

The two transitions with only 'this:alicepays()' or 'this:bobpays()' exist because giving a trinket requires payment, but payment doesn't require giving. When a transition needs to be able to fire without a downlink calling its uplink, we add a transition without inputs or outputs (which makes it always enabled) that has a downlink calling that uplink.

For mutual exclusion of activities, we add a place with a token and connect it to the transitions using reserve arcs. If the transitions are in different nets, we add the place to each, add a transition to each, and then use up- and downlinks synchronise those with the original transitions in the other nets.

[3] Inheritance is nothing more than a way to avoid drawing the same thing multiple times. Metarelations to inheritance could be interesting, but aren't part of the current version of CRML

[4] In a future version of CRML, it will be required that relations providing to slots are blocking, instead of making them block implicitly.

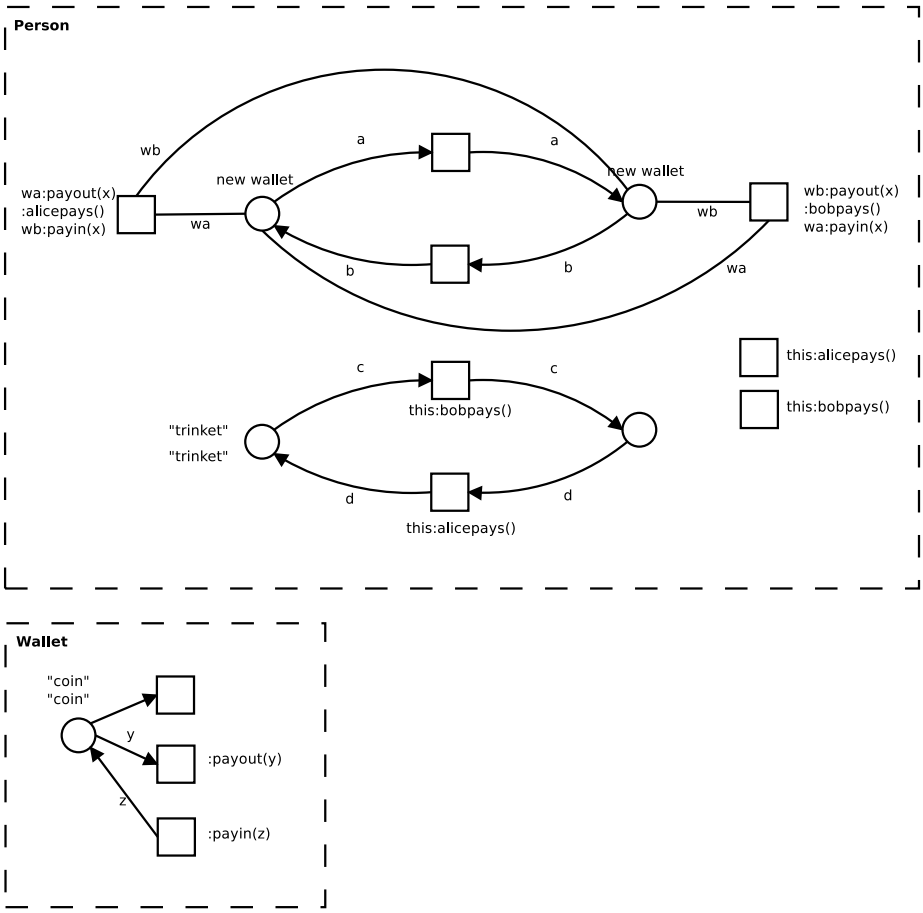

Figure 2.5: The RENEW translation

2.3 Advanced Preparation for Translation

CRML has some features that appear to be unusual but which are equivalent to more convoluted constructions that are possible both in CRML and in other languages. The transformations that remove the use of those features from a diagram are described below. They should be executed in the order they are described.

2.3.1 Metarelations

For every relation that has relations pointing to it or that has inherent attributes, we add an empty object description at the same depth of object description as the least deep nested one affected by the relation. All relations from and to the relation being considered are moved to this new object description. Such an object description is called a stand-in.

If the relation is a requirement, it will be replaced by a forked requirement: *requires*/*forbids* (target of relation) *or forbids* (stand-in).

If the relation is state modifying, the diagram element[5] that does the providing or revoking will be duplicated (with two different names). One will require the stand-in and have the relation being considered, the other will forbid the new slot and won't have the relation. Any requirements to the duplicated element will be connected to both copies through an *or*-fork. Any state modifying relations to the duplicated element will require or revoke both.

2.3.2 Inherent Attributes

State Modifying Inherent Attributes

If an object description or stack provides something as an inherent attribute, any activity, object description or stack that revokes the element provided by the inherent attribute will now forbid the providing element. Each such revoking element is copied, with all its relations except the *revokes* connected to the provided element and the new *forbids*. These copies will require the providing element.

Similarly, if an object description or stack revokes something as an inherent attribute, any activity, object description or stack that provides the element revoked by the inherent attribute will now forbid the revoking element. Each such providing element is copied, with all its relations except the *provides* connected to the revoked element and the new *forbids*. These copies will require the revoking element.

Inherent Requirements

If an object description or stack requires something as an inherent attribute, any activity that revokes the required element will now also revoke the requiring element, and any activity that provides the required element will now also provide the requiring element.

Similarly, if an object description or stack forbids something as an inherent attribute, any activity that provides the forbidden element will now revoke the forbidding element, and any activity that revokes the forbidden element will now provide the forbidding element.

[5] If this also is a relation, this one must be transformed first. Since there is no practical use for inherent-*provides*-loops, that will always be possible.

In case of *or*-forked inherent requirements, a copy needs to be made of each activity the relations of which are changed as described above. One will require that the part of the requirement not affected by it is satisfied, the other will forbid the same part of the requirement. One will modify the state as described above, the other won't, such that the behaviour of the whole system remains the same.

2.3.3 Provided or Revoked Activities

If an activity is provided or revoked, a stand-in in the inventory of the object description that has the activity is provided or revoked instead. This new object description is then required by the activity that was provided or revoked.

2.3.4 Requirements

Location Requirements

When an activity is drawn in a nested object description rather than in one at the top level, that implies it can only be performed when the object is in that location. Usually the activity interacts with diagram elements nearby[6], and we don't need to anything to make it impossible to perform the activity in other locations.

In the unusual case, a stand-in is added to the object description and required instead of the location requirement. The stand-in is provided by all activities that put the object in the location, and revoked by all that remove it from the location. If the object is in the location in the initial state, a copy of the stand-in without providing relations pointing at it needs to be drawn.

Forked Requirements

An activity with forked requirements is replaced by multiple activities that:

- all forbid each other.
- all have different sets of unforked requirements that match a condition that satisfies the original forked requirements. There must be an activity for each possible combination.

[6] If it doesn't, something important is likely missing from the model.

Chapter 3

Conclusions

Comparing [Bui10] to the first chapter of this report will show the many small improvements and clarifications that have been made. For the purpose of improving the definition of the semantics of CRML, the creation of the translation method is a success.

On the other hand, the necessity to make diagrams explicit is very limiting. To prove something about a CRML diagram, we need to prove it for all possible worlds, not for a specific one.

Another problem is that although CRML has a lot of similarity with Petri nets, the devil is in the details. CRML is such a rich language that it is unlikely there exists a Petri net dialect that all possible combinations of CRML elements can be translated to. Without such a Petri net dialect, reusing tools for and knowledge about Petri nets for CRML is impractical.

3.1 Advantages and Disadvantages of CRML

CRML is based on constraints, it requires the reader of a diagram to imagine what would happen when the constraints are applied to actual objects. Because reading a CRML diagram requires work, it may be perceived as a difficult language.

On the other hand, a modelling language that describes behaviour without any hint of the reasons why that behaviour is the way it is requires the reader to imagine those reasons – which may lead to them drawing incorrect conclusions.

Although CRML describes both structure and behaviour, it will often not be suitable to be used as the only modelling language in a document. A good explanation of a concept requires a spectrum of perspectives, with examples being one extreme, and rules (constraints) being the other.

The emphasis on constraints is natural for copyright licenses. Although constraints are everywhere, it remains to be seen whether users in other domains will consider the method of thought used in CRML useful.

3.2 Future Work

The direction of future work on CRML will be influenced by who is interested in the language and what they wish to use it for. Further investigation of the possibilities for using CRML in cell biology may happen, but it is equally likely someone will want to use the language for a different and unexpected purpose.

Some ideas for improvements to CRML are mentioned in this paper. CRML will get a formal definition after the informal definition stops changing. The correctness of the method to translate to Petri nets can then be verified, and implemented in software (this does require that CRML elements are added to diagram drawing software). However, because of the limitation to explicit diagrams it will likely be more useful to use a custom method to prove properties of CRML diagrams, or to develop a translation method to a modelling language that is not based on Petri nets.

Appendix A

Examples

A.1 Creative Commons Licenses

The Creative Commons Licenses are a popular set of permissive copyright licenses. Creative Commons has created a specification for describing their and other licenses in RDF, called ccREL[1] [AALY08]. CcREL uses the verbs *permits* (equivalent to *provides* in CRML), *prohibits* (equivalent to *forbids*) and *requires* (the same as in CRML). *Revokes* is missing. The object of each verb is a single word, the meaning of which is not expressed in the RDF like it would be in a CRML diagram.

The models in this example do not describe every use case mentioned in the licenses, but, excluding anything related to "noncommercial" use, it is most likely possible to create complete models.

Each of the licenses contain some of the following terms:

- Attribution (BY)
- Noncommercial (NC)
- No Derivative works (ND)
- Share Alike (SA)

The licenses are:

- BY
- BY-NC
- BY-ND
- BY-SA
- BY-NC-ND
- BY-NC-SA

All of these licenses *provide* the right to distribute copies of a work (see figure A.1), and except for the ND licenses they also provide the right to distribute derivative works (see figure A.2) beyond what is allowed by copyright law as fair use. Both rights are only provided when certain restrictions (*requirements*) are satisfied.

[1] Creative Commons Rights Expression Language

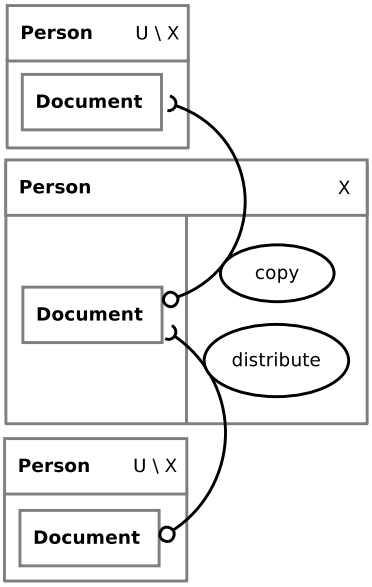

Figure A.1: Distributing copies

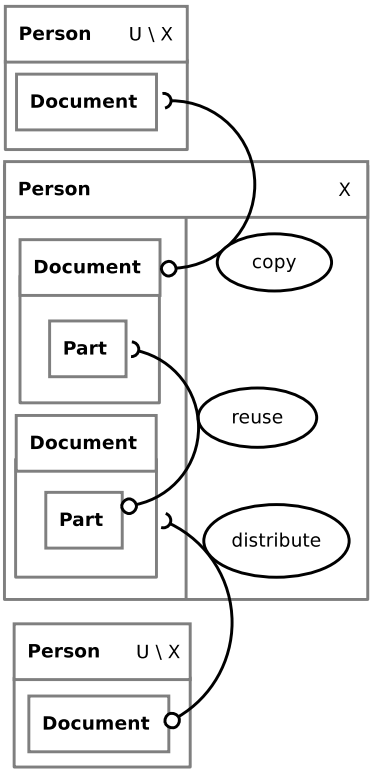

Figure A.2: Distributing derivative works

A.1.1 Requirements

NC: Noncommercial

The noncommercial requirement is defined in section 4.b of [Crea]:

> You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

We might model this as providing activities under the condition that those activities don't require movement of money-objects, but this would be incorrect: NC-licenses are, for example, believed to allow a work to be used for fund raising for a nonprofit organisation. It is not possible to model ambiguous terms such as "primarily intended", "commercial advantage" or "private", and there is controversy about what they mean[Creb].

BY: Attribution

The BY requirement means the license must be included in copies and derivative works, And that attribution must be given to the original author (authors of derivative works may in turn also demand attribution). It is expressed in figure A.3. Including the red part of the diagram makes it a description of a BY license, without the red part it is a BY-ND license.

Note the license has *inherent attributes*: activities are possible because the license exists.

SA: Share Alike

The BY-SA license in figure A.4 is like the BY license, except it requires that parts of a derived work not created by the original author must be published using the same license.

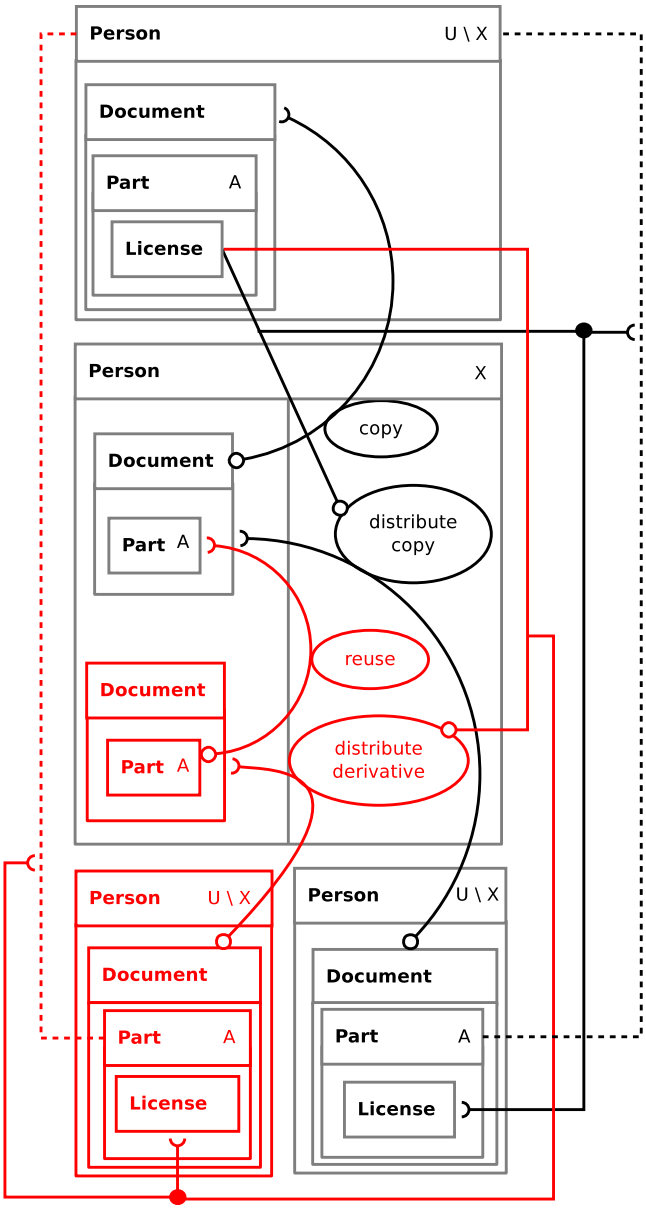

Figure A.3: BY and BY-ND licenses

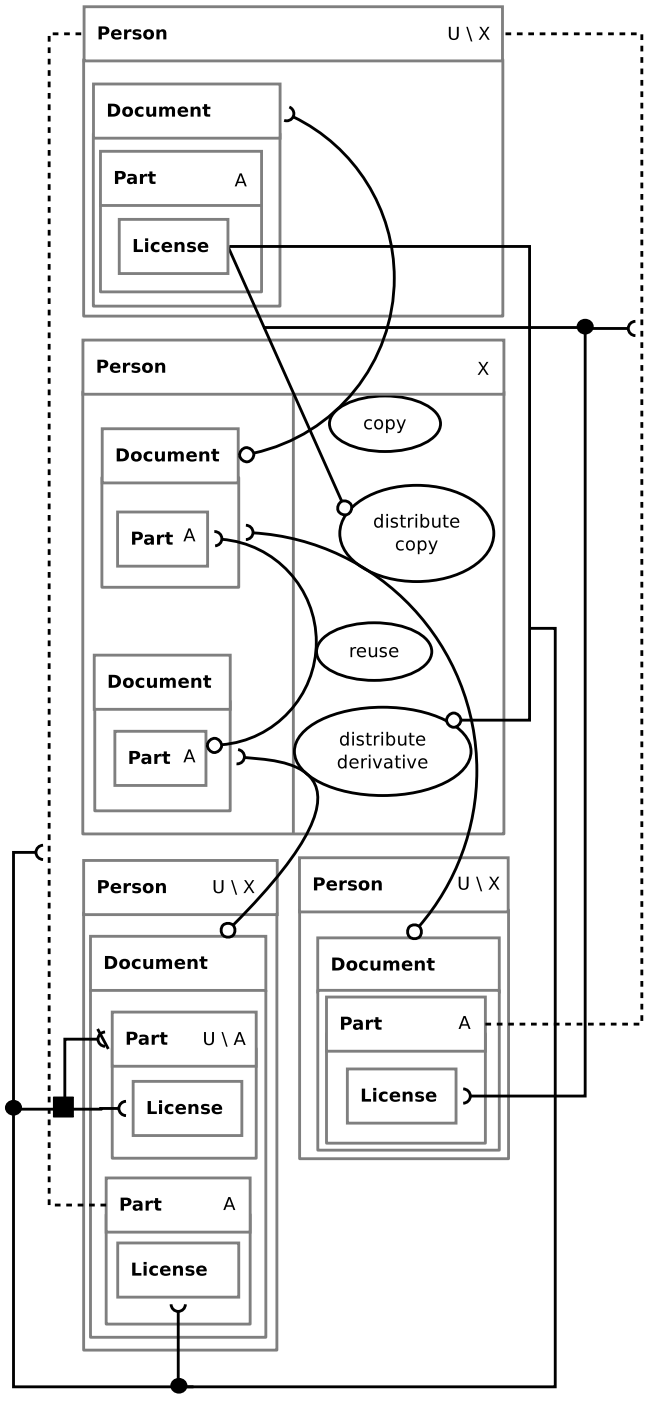

Figure A.4: The BY-SA license

A.2 A Neuron

Figure A.5 shows an incomplete but hopefully mostly correct[2] model of a neuron, based on the information on pages 411—419 of [Alb04]. The complete model would not fit on a page, however it could have been split into multiple diagrams without changing its meaning. For this example the incomplete model is sufficient.

The neuron is divided in sections to model the propagation of a signal along the axon. Section 0 is a dendrite (not strictly part of the axon, but this is a simplified model...).

Each section contains at least one Na^+ channel, which is shared with whatever is outside the cell.

A Na^+ channel has two inventory items to model its state: *IsClosed* and *IsInactive*. A Na^+ channel starts in the closed state, as is shown by the *IonChannel* object description containing only an *IsClosed*.

A Na^+ channel opens when the membrane potential is less negative than usual, which here is modelled by a positive charge of *m* in the previous section. When it is open, Na^+ ions (which are positive) flow in through it until their concentration inside and outside the cell is similar (the difference should be below a certain threshold, for the model near-equality is assumed). The channel becomes inactive and may then close. The way Na^+ (and other) ions are removed from the cell is not shown in the diagram – it is inherited from general cell behaviour.

The presence of negative ions also affects the membrane potential. We can pretend they form neutral salts with the positive ions and by doing so reduce the number of positive ions. This behaviour is partially shown for clarity, but really also belongs in a description of the class *Cell*. The split-activities that are the opposites of the combine-activities have not been drawn.

Ca^{2+} channels work similar to Na^+ channels, but let in a different ion.

In response to the presence of Ca^{2+}, vesicles merge with the cell membrane (here that's modelled as self-destruction) to release a neurotransmitter.

Neurotransmitters bind to special ion channels in the dendrites. When a neurotransmitter binds to an inactive channel, the result is an active channel, which, depending on its type (and the matching neurotransmitter) lets in positive or negative ions. Only one type of transmitter gated channel has been drawn. A more complete model should at least also contain one that lets in negative ions. The models for other types of transmitter gated channels are identical except for the names of the channels, the neurotransmitters and the ions.

[2] I'm not sure whether Ca^{2+} ions could appear in all the places they have been drawn.

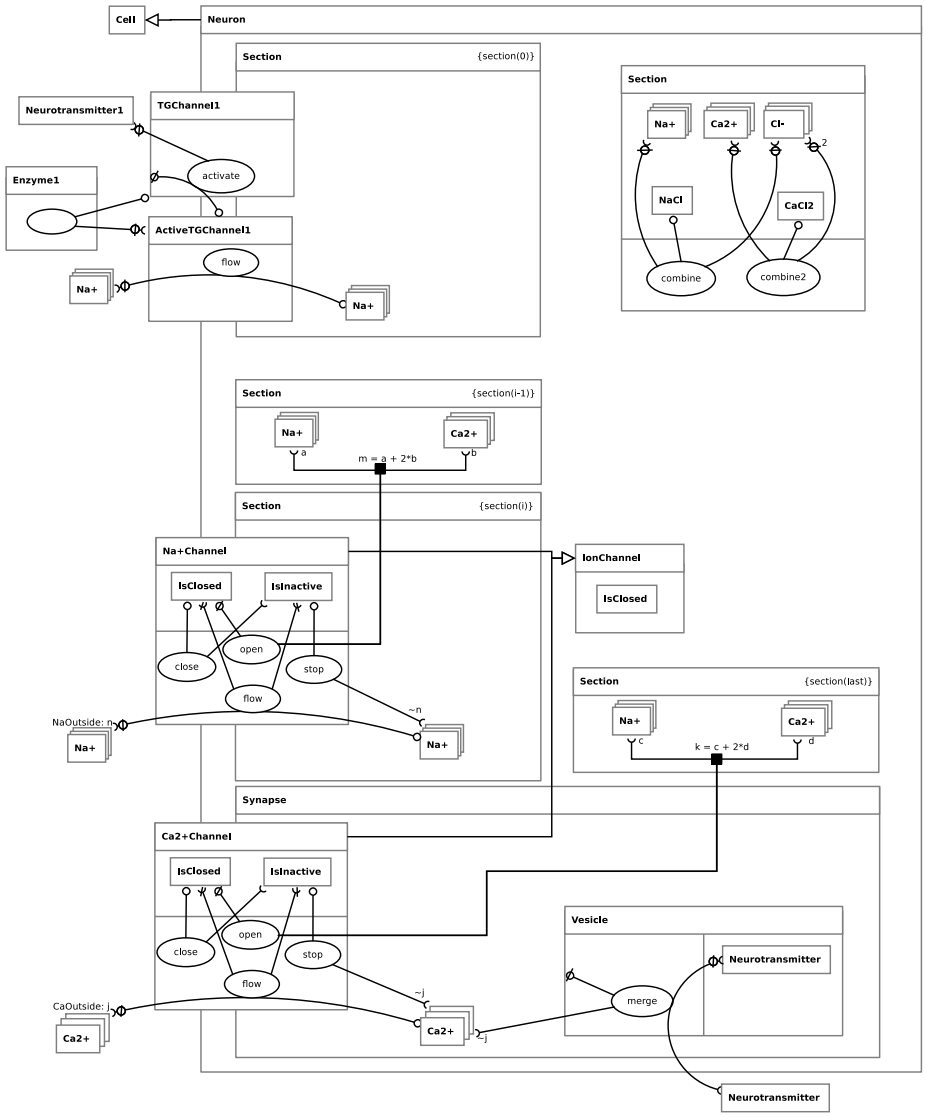

Figure A.5: Model of a Neuron

Glossary

acting on behalf of An object inside the inventory of another performs an activity. 9

activity A behaviour an object may perform. 9

activity-state relation A state modifying relation or a requirement. 11

actor description An object description with activities. 9

actual object An object that is part of what is modelled, rather than part of the model. For example a real world object or an object that is part of a computer program. 7

explicit diagram A diagram that describes its entire world: it does not contain stacks with unspecified initial heights, and all object descriptions are either assigned sets without free variables or have a providing relation pointing at them. 23

false friend In linguistics, a pair of words, phrases or symbols in two languages that look or sound similar, but differ significantly in meaning. 7

free variable A symbol that doesn't refer to a specific object or set but can be replaced by a reference to one. 8

guard An expression that must evaluate to true if the program execution is to continue in the branch to which it is applied. 6

inherent attribute A relation between the activity of existing of an object and something else. 13

inventory The structure of an object, things it has. 9

metarelation A relation to a relation. 5

model checking Automated verification that a model matches a certain specification, e.g. that the system modelled cannot reach an unwanted state. 6

multiplicity The number of actual objects affected by a side of a relation. 17

named stack A stack that also is a slot. 17

passive object description An object description without activities. 9

requirement A relation that shows what is necessary for an activity to be possible. 11

shared inventory An object description that overlaps the inventory of more than one other object description. 11

slot A labelled object description inside the inventory of another object description. 16

stack An object description for which the number of actual objects is counted. 17

stand-in An object description that receives the relations that were previously connected to some other type of diagram element. 31

state modifying relation A relation that creates or destroys an object or a relation. 11

universe The set of all objects belonging to a certain class and its subclasses. 7

visual programming Creating programs by drawing diagrams rather than writing text. 6

world A set of actual objects that the rules in a CRML diagram can be applied to. 7

Bibliography

[AALY08] Hal Abelson, Ben Adida, Mike Linksvayer, and Nathan Yergler. ccREL*: The Creative Commons Rights Expression Language*. http://wiki.creativecommons.org/CcREL, http://wiki.creativecommons.org/images/d/d6/Ccrel-1.0.pdf, retrieved March 2014, 2008.

[Alb04] Alberts et al. *Essential Cell Biology*. Garland Science, second edition, 2004.

[Bui08] T Buitenhuis. *Modeling the Passiflora system*, 2008.

[Bui10] T Buitenhuis. *CRML: a Constraint Relation Modeling Language*. http://crml.beigeserver.eu/2010/crml.pdf, retrieved March 2014, 2010.

[Crea] Creative Commons. *Attribution-NonCommercial 3.0 Unported*. http://creativecommons.org/licenses/by-nc/3.0/legalcode, retrieved March 2014.

[Creb] Creative Commons. *Defining Noncommercial*. http://creativecommons.org/weblog/entry/17127, http://wiki.creativecommons.org/Defining_Noncommercial, retrieved March 2014.

[GNO] GNOME. *Dia*. https://wiki.gnome.org/Apps/Dia, retrieved March 2014.

[KWDC09] Olaf Kummer, Frank Wienberg, Michael Duvigneau, and Lawrence Cabac. *Renew – User Guide*. http://www.renew.de, http://www.informatik.uni-hamburg.de/TGI/renew/renew.pdf, retrieved August 2010, newer version at same address in March 2014, 2009.

[Obj] Object Management Group. *Unified Modeling Language*. http://www.uml.org, retrieved March 2014.

[Val04] Rüdiger Valk. Object petri nets. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 819–848. Springer Berlin / Heidelberg, 2004.

[Wol] Wolfram Research. *Mathematica Tutorial*.
http://reference.wolfram.com/mathematica/tutorial/PuttingConstraintsOnPatterns.html,
retrieved March 2014.