Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 259 (2009) 87–104
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Diagrammatic Formal Specification of a Configuration Control Platform

John Howse[1] , Steve Schuman, Gem Stapleton[2]

University of Brighton, UK

Ian Oliver[3]

Nokia Research, Finland

Abstract

This paper presents a diagrammatic logic framework that is suitable for use in formal specification and for reasoning about and refining formal software models. We take a case study style approach to presenting the framework by developing, in some detail, an abstract model for a transparent configuration control platform. The model is built up by stages, corresponding to separate concerns of configuration control. Each successive level is a refinement of the previous level. We discuss the possibilities for developing tools to support the use of the diagrammatic logic, including automated diagram drawing and reasoning procedures. Our wider goal is to make a formal specification easier for its clients to understand.

Keywords: configuration control case-study, constraint diagrams, object-oriented formal specification, visual modeling, visual refinement.

1 Introduction

Software specifications are typically presented in symbolic notations or with (at best) semi-formal diagrammatic notations such as those found within UML. Formal reasoning about specifications is almost exclusively performed with symbolic notations. Many people find symbolic notations inaccessible and hard to use. Added to that, specification construction, conceptualization and refinement can be difficult, and is hindered by the inaccessibility of the syntax available to the user. The provision of a fit-for-purpose, more widely accessible notation specifically designed for formal specification and reasoning may be helpful to a large community of users. Diagrammatic notations are potentially a viable alternative to symbolic notations.

[1] Email: John.Howse@bton.ac.uk

[2] Email: g.s.stapleton@bton.ac.uk

[3] Email: Ian.Oliver@nokia.com

doi:10.1016/j.entcs.2009.12.019

Moreover, we argue that using diagrammatic notations for reasoning, in addition to specification, can bring huge benefits. It is important to reason about specifications, since this leads to a better understanding and can reveal unintended consequences.

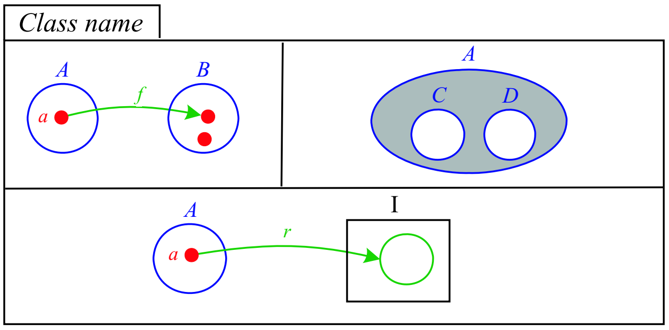
Configuration control has long been recognized as playing a vital, even critical part in the software production process. Indeed, several ever-more sophisticated platforms are now coming on the market to support this process. Almost all of them attempt to be 'transparent', by providing a namespace that is compatible with the use of standard development tools. In this paper we develop an abstract model for such a configuration control platform. It illustrates how the visual formalism may be used in order to build up a precise specification.

A previous case-study [10] presented a much simpler example. We argued there that a diagrammatic formal specification should be easier for any client to understand, enabling them to play an active role in the requirements engineering process [20]. The specification considered here has far more complex constraints, but these will be well-understood by software professionals – who are its ultimate clients. This case-study is likely to be of interest to anyone with similar expertise.

In this paper, we propose a diagrammatic framework for specification and refinement. The notation builds on previous work in the diagrams area. In particular, the framework uses constraint diagrams [13] as a basis, but we extend and modify the syntax. Moreover, our semantic interpretation of this notation is not the same as the semantics of constraint diagrams [5,21]. We assign semantics that are appropriate for their use in specification and refinement. The framework also builds on the object-oriented formal specification framework [16,17]. The most distinctive convention of this framework is known as "the rest stays unchanged" [15]: it allows post-conditions of operations to be expressed in a weaker or minimal form – which significantly simplifies usage in practice.

2 The Diagrammatic Framework

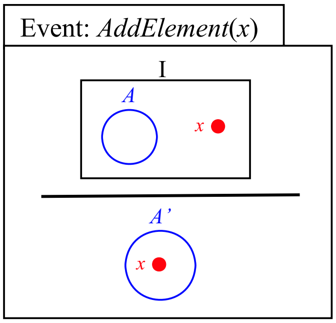
A class is modeled in terms of an invariant and its operations. The following diagram is an example of an invariant:



The diagram consists of three sub-diagrams. The closed curves (circles, ellipses, rectangles) represent sets. In each sub-diagram, the curves form an Euler diagram and their spatial relationships express semantic relationships between the sets: nonoverlapping curves assert that the sets are disjoint; a curve placed inside another asserts a subset relationship. We use the convention that labelled rectangles represent types. The dots are called *spiders*. Unlabeled spiders assert the existence of elements in the sets represented by the regions of the diagram in which they are placed. The labeled spiders in this diagram are acting as free variables. An *arrow* represents a binary relation, where its *source* and *target* may be either a spider or a curve; its target then corresponds to the image of its source under a relation identified by the label on that arrow.

In the first sub-diagram there are two disjoint sets A and B. The spider labeled a is a free variable and is the source of the arrow f, while the target of f is a spider in B; thus f represents a function mapping each element of A to an element in B. Different spiders represent distinct elements, so there is an element in B that is not the image of a under f. In the second sub-diagram C and D are disjoint sets and each is a subset of A. As there are no spiders in it, the shaded region represents the empty set. So this sub-diagram asserts that A is partitioned into subsets C and D. In the third sub-diagram the rectangle represent a type; we use names from a distinguished alphabet $\mathbb{A}$,...,$\mathbb{Z}$ to define given (fully-abstract) sets, taken as type-parameters of that class. The arrow labeled r represents a relation as it maps each element of A to a subset of $\mathbb{I}$. The sub-diagrams are conjoined.

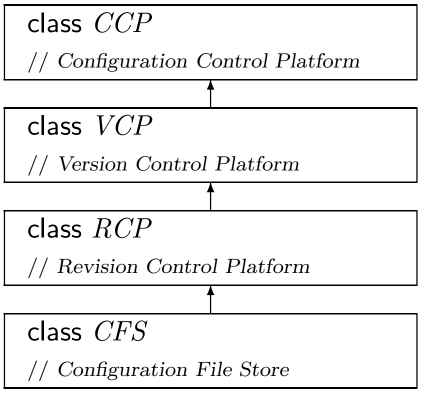
Operations are divided into queries and events. Queries specify operations that may be applied to an object of a class leaving its state unchanged, whereas events specify their allowable changes-of-state. Either may have input- or output-arguments that are declared in its pre-condition. Each event also has a post-condition, wherein 'dashed' names denote values of the corresponding variable after any occurrence of the event.



This event is specified in terms of a pre-condition and a post-condition. The pre-condition is specified above the line and the post-condition below the line. The diagram can be interpreted as 'if the conditions above the line hold then the conditions below the line hold after an occurrence of the event'. In the pre-condition x (which is of type $\mathbb{I}$) is not in A and in the post-condition x is in A', the updated version of A. So this event adds x to A.

3 The Abstract Model

We will use refinement to build up our model as simply and abstractly as possible. The four successive stages of its overall structure are depicted below:

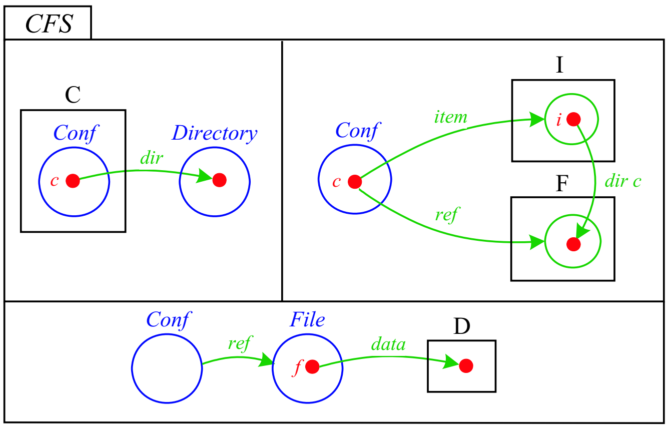

class CCP
// Configuration Control Platform
class VCP
// Version Control Platform
class RCP
// Revision Control Platform
class CFS
// Configuration File Store

The modeling process will develop each of the classes in turn, with class RCP refining class CFS and so forth. Each of the four classes will be presented and defined diagrammatically, highlighting the refinement of the model clearly. Every level reflects a separate concern or requirement of configuration control. We will document all parts of its formal specification by an informal description. Each level will be specified in terms of an invariant and operations on the class.

3.1 Configuration File Store

We start by considering a simple "Configuration File Store", formally specified as class CFS:

A Configuration File Store maintains a set of *configurations* (uniquely identified by an element from type $\mathbb{C}$). Every configuration has its own *directory*: a finite set of *items* (distinct names from type $\mathbb{I}$), each of which refers to a particular *file* (via *references* from type $\mathbb{F}$). The same file may be *shared* by several *different* configurations. The set of stored files then comprises all those referred to by at least one identified configuration. The type $\mathbb{D}$ of *data* in such files is also abstract.



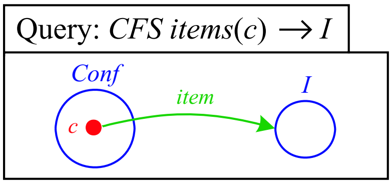
The diagram specifying the invariant of class *CFS* consists of three sub-diagrams. The first diagram asserts that there is a set *Conf* (of configurations) which is a subset of abstract set $\mathbb{C}$. Furthermore, each element of *Conf* is associated with a directory by the function *dir*. The second diagram asserts that each element c of *Conf* is associated with a set of items and a set of references by the relations *item* and *ref*, respectively. Each item associated with c is linked to a reference associated with c by the function *dir* c (indicating that each element of the set *Directory* of the first diagram is in fact a function). The third diagram asserts that the codomain of the relation *ref* is the set *File* and each element of *File* is associated with an element (its data) of the abstract set $\mathbb{D}$ by the function *data*.

The key insight in the specification of a Configuration File Store is that separate *configurations* may be modeled as ‘flat’ directories in a conventional file-system (users of which are presumably subject to the usual ‘access controls’). Items in a directory are not further constrained: they may be source-code, test-data or even specifications. The *CFS* invariant suggests (but does not impose) using a ‘reference-count’ strategy to ensure that all files referred to will be *maintained.* Any platform for configuration control must meet this fundamental requirement.

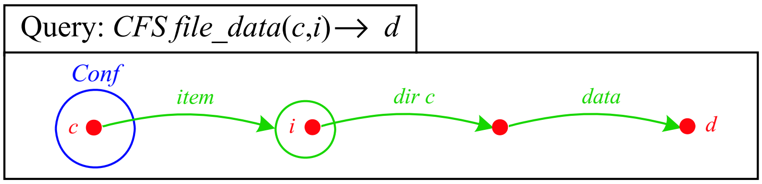
Queries of class CFS.

The simplest queries provided at this level apply to only one identified configuration:

The query *items*(c) shows the set I of existing item-names for a configuration c.

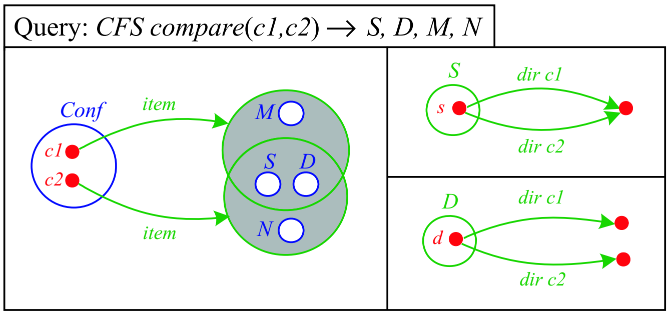


The query *fileData*(c,i) shows the data d stored on the file for item i in configuration c:



In these diagrams the labeled spiders represent the parameters of the queries. A further query, applying to two *different* configurations, may prove particularly useful in the context of this application:

The query compare($c1$,$c2$) compares separate configurations $c1$ and $c2$ by splitting their item-names into four disjoint subsets: common names S that refer to the same file; common names D that refer to different files; the names M present in $c1$ but not in $c2$; and the names N present in $c2$ but not in $c1$:

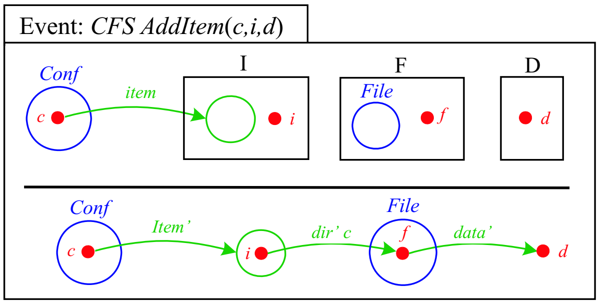


In this diagram, shading asserts that the represented set is empty and two different spiders represent distinct elements, so $c1$ and $c2$ are distinct.

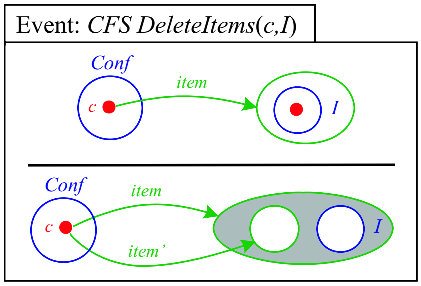
Events of class CFS.

An event, a state-changing operation, is specified in terms of pre-conditions and post-conditions. In the diagrammatic representation the pre-condition is specified above the line and the post-condition below the line. The diagram can be interpreted as 'if the conditions above the line hold before the event then the conditions below the line hold after the event'.

The event *AddItem*(c,i,d) adds a newly-created item i, which refers to a newly-created file with data d, to configuration c, provided its name is distinct:

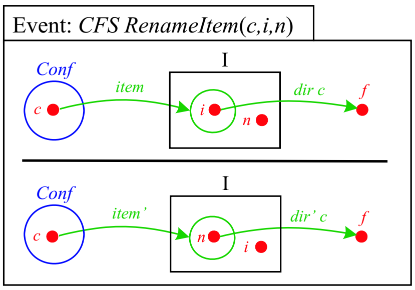


The event *DeleteItems*(c, I) deletes a non-empty subset I of its existing items (along with their references) from configuration c:



Further events might well be required – partly to improve *ease-of-use*, but also to encourage *sharing* of files and thus avoid unwanted copies – e.g. the following.

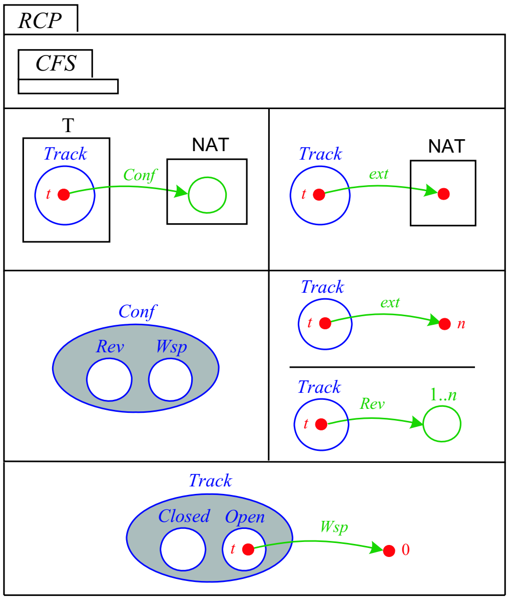
The event *RenameItem*(c,i,n) renames item i, in configuration c, as n (whilst still referring to the same file), provided this new name is distinct:



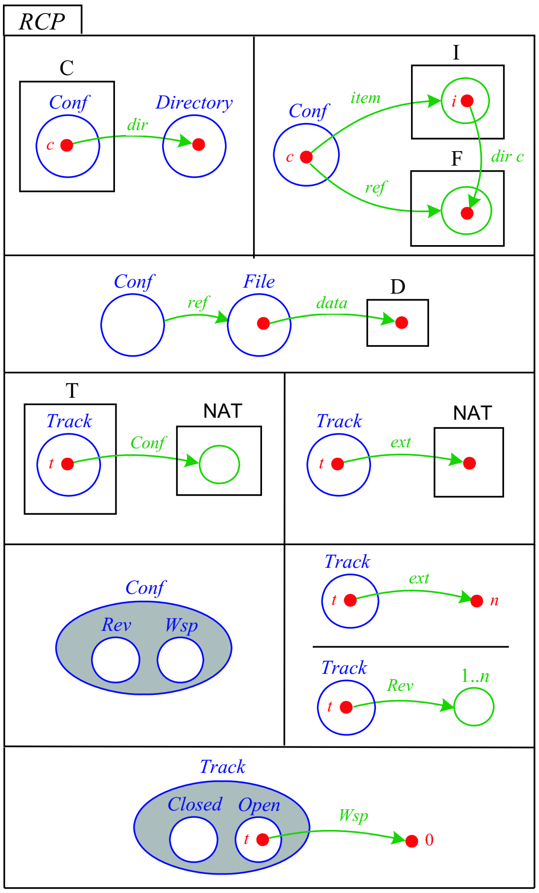
3.2 Revision Control Platform

We now refine class *CFS*, by strengthening its invariant, adapting some of its operations, and defining some new operations, to specify a simple "Revision Control Platform" as class *RCP*:

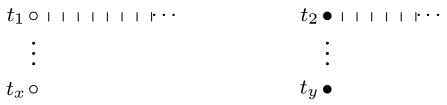
A Revision Control Platform extends a Configuration File Store to define a finite, initially-empty set of *tracks* (having unique names from type $\mathbb{T}$). Each track has an *extent*: its current number of consecutive positions. This gives rise to a *namespace* consisting of the known track-position *pairs*. All positions > 0 identify *read-only* configurations corresponding to successive *revisions* (so such identifiers for the same track are linearly-ordered). A track may also have an associated *workspace*: a transient, *modifiable* configuration (always at position 0). Whenever its workspace exists, that track is *open*; otherwise, it is *closed*:



We include the specification of class *CFS* within the specification of class *RCP*. Thus class *CFS* has been refined by strengthening its invariant to produce class *RCP*. NAT is the type of Natural Numbers. A configuration is now modeled as a pair: a track and its position (a natural number). The middle right-hand sub-diagram is an 'if ... then' statement as discussed earlier. The label $1..n$ represents the set $\{1, 2, \ldots, n\}$. The *expanded* version of *RCP*, in which all diagrams are shown explicitly, is:



A Revision Control Platform might be visualized as some set of uniquely-named *tracks*, each of which identifies its zero or more successive *revisions*. Independently, every track may be either *open* (○) or *closed* (●). When open, its associated *workspace* may be seen as the next (but not yet committed) revision within that track:



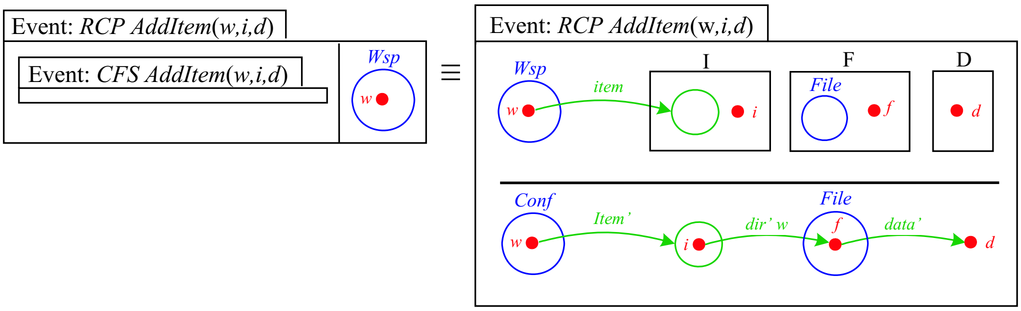
From this level, associated operations will be divided into two different groups: *Configuration-Based Operations* that apply to the individual configurations which are identified, and *Track-Management Operations* that manage the separate tracks within this overall namespace.

Configuration-Based Operations of class RCP.

All queries in this group are simply promoted from *CFS*, so their descriptions are not repeated. These may be applied either to a known revision or to an open workspace.

Events in this group must be applied to an open workspace – ensuring that they are the only *modifiable* configurations. The events of *CFS* are suitably refined so that this is the case – for example, the *CFS* event *AddItem*(c,i,d) is refined to the *RCP* event *AddItem*(w,i,d).

The new item i referring to a newly-created file with data d can be *added* into workspace w, provided its name is distinct:

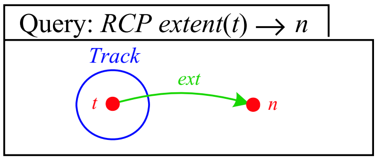

Event: RCP AddItem(w,i,d)
Event: CFS AddItem(w,i,d)
Wsp
w
≡
Event: RCP AddItem(w,i,d)
Wsp
w
item
I
i
F
File
f
D
d
Conf
w
Item'
i
dir' w
File
f
data'
d

The specification of *CFSAddItem* is included the specification of *RCPAddItem*. The pre-condition of *CFSAddItem* is conjoined with the sub-diagram to ensure that the event is applied to an open workspace. An expanded version of this event is also shown.

Track-Management Operations of class RCP.

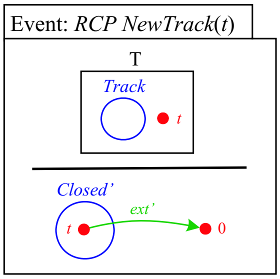
Only a few simple queries are likely to be required at this level. For example:

The query *extent*(t) shows the current extent n, i.e. its number of known revisions, for any track t:

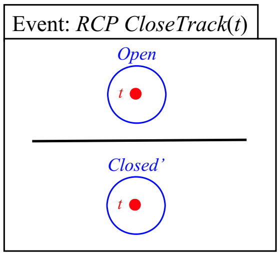


Its track-management events are also relatively simple and include the following.

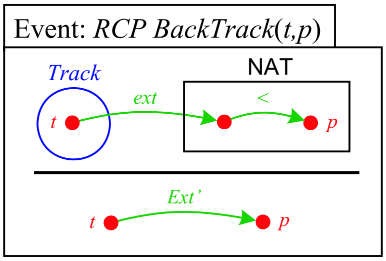
The event *NewTrack*(t) defines a new track t, provided its name is unique; it is initially closed, with no revisions:



The event *CloseTrack*(t) closes an open track t, so that its workspace no longer exists:



The event *BackTrack*(t) resets the extent of track t to earlier position p so that its set of all later revisions no longer exist:

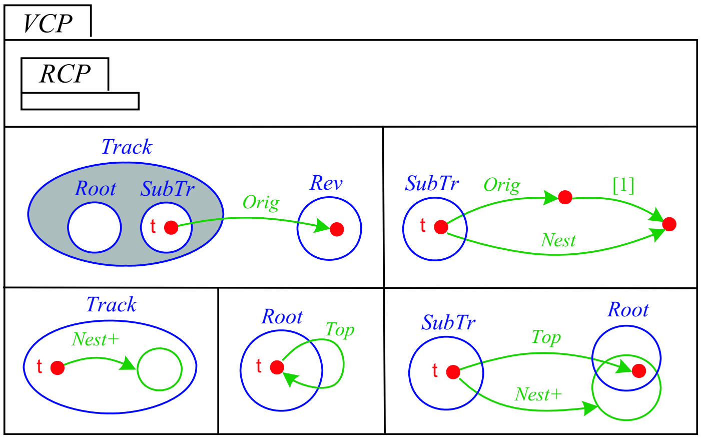


All development occurs only in an open workspace, which starts from empty or a copy of some revision r. Copies of this workspace can be 'saved' as its incremental revisions, until that track is closed. Thus, 'open'/'close' are analogous to the traditional 'check-out'/'check-in' operations for configuration control. Other events serve to manage that track and its number of known revisions.

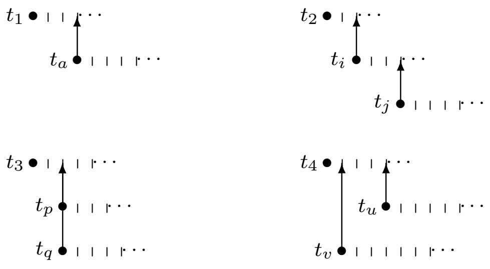
3.3 Version Control Platform

We refine class *RCP*, to support 'variant versions' as well as simple revisions. Such a "Version-Control Platform" is specified here as class *VCP*.

A Version Control Platform extends a Revision Control Platform so that certain tracks may be defined with some revision of a different track as their *origin*, whence they are said to be *nested subtracks*; tracks with no such origin are taken to be top-level *roots*. Every root, with its branching subtracks if any, must always form a *tree*: this partial-ordering means that all of those revisions may be seen as variant versions of one another. Each track then has a root as its *top* track:



Possible tree-structures may be depicted as follows (using vertical arrows to indicate subtrack origins):



The existence of subtracks means they too can be developed *in parallel* (imagine for example that the tracks or subtracks shown above are all open at once, so any of them may then have more revisions). Allowing such potentially 'divergent' developments to co-exist is often seen as one problem that configuration control should always rule out. In the end however, this is just another *trade-off*...

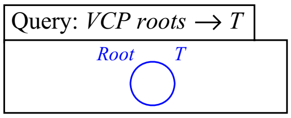
Configuration-Based Operations of class VCP.

These are all independent of the subtrack structuring introduced at this level, so such queries and events could be directly promoted from *RCP*. But we shall defer any promotions until our final level, where other constraints will also be apparent.

Track-Management Operations of class VCP.

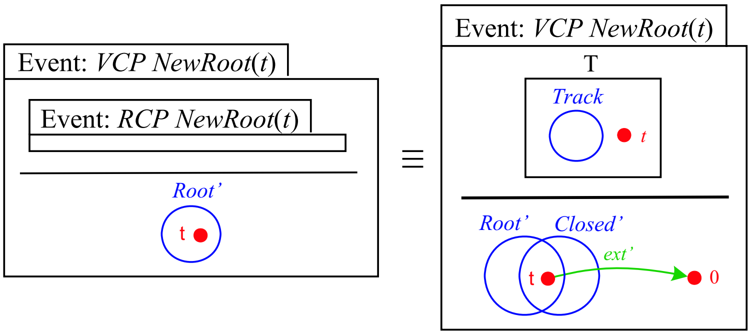
To take subtrack structures into account, some new queries will now be required, for example the following.

The query *Roots*(t) shows the set T of all top-level roots:



A few refinements of *RCP* events will be required, for example the following.

The event *NewRoot*(t) defines a new root t, provided its name is unique; it is initially closed, with no revisions:

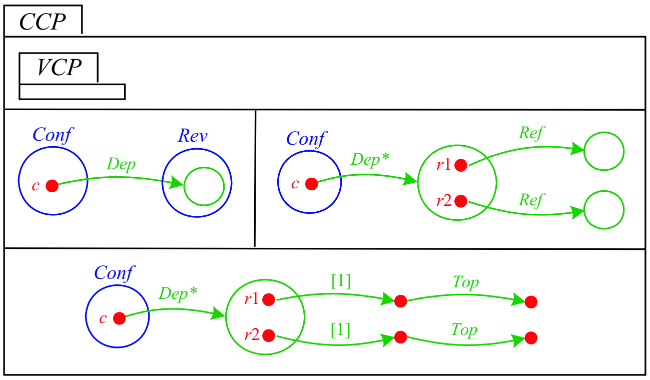


Further track-management operations may be required at this level, to *explore* or even *re-arrange* its nesting structure. But, in the interests of simplicity, we will not pursue such possibilities here.

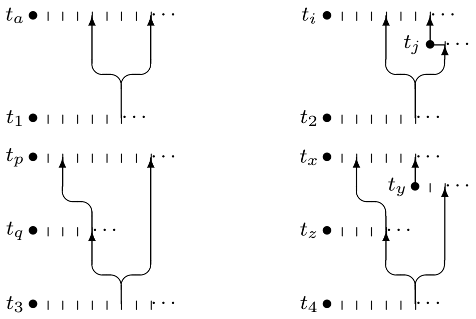
3.4 Configuration Control Platform

We now refine class *VCP*, to define our final "Configuration Control Platform" specified as class *CCP*. It is motivated by observing that *dependencies* represent yet another source of complexity in software development, which might usefully be put under configuration control.

A Configuration Control Platform extends a Version Control Platform to record direct *dependencies* for every identified configuration: i.e. that set of related revisions which would need to be *transitively* imported into some surrounding context to complete its definition. No configuration may depend, directly or indirectly, on itself; furthermore, it and all of its existing dependencies must have *different* top tracks as well as refer to *disjoint* files:



Here *Dep** is the transitive closure of *Dep*. The *different top track* constraint avoids any so-called 'version skew'. This may be depicted (using curved arrows to indicate dependencies) as cases where one configuration depends, directly or indirectly, on at least two configurations which are defined as variant versions of each another:

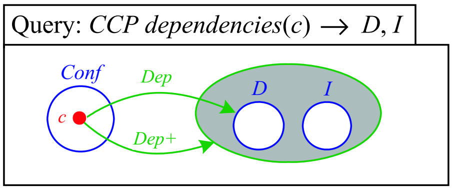


The *disjoint file* constraint may be seen as extending to all dependencies the constraint on configurations that items in any directory must always refer to different files. We note here that these constraints further reduce the need for (and dangers of) *nested* directories in software development – reinforcing our initial abstraction based on 'flat' ones.

Configuration-Based Operations of class CCP.

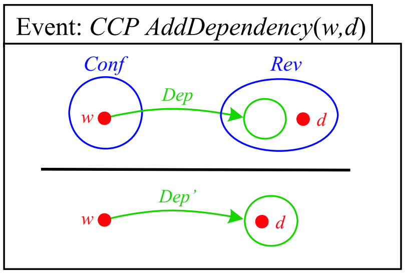
Some new queries will now be required in this context, for example the following.

The query *dependencies*(c) shows the dependencies for configuration c; these are divided into its *direct* ones D and *indirect* ones I:



Other such queries are directly promoted from *CFS*. New events are required to *modify* any dependencies. As before, these may only be applied to some open workspace. For example:

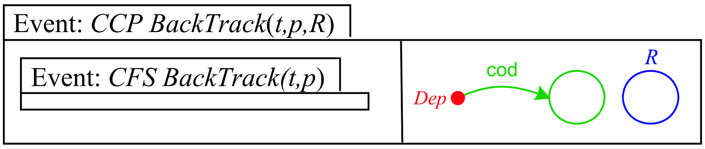
The event *AddDependency*(w,d) adds a new dependency d to workspace w, provided it and all its existing dependencies have top tracks which differ from that of w itself or any existing dependency and they all refer to disjoint files:



Other events are directly promoted from RCP.

Track-Management Operations of class CCP.
Here, all such queries are promotions from lower levels. Some events must be further refined, to deal with dependencies. For example:

The event *BackTrack*(t,p,R) resets the extent of track t to earlier position p so that its set R of all later revisions no longer exist, provided R includes neither any subtrack origins nor any dependencies for other configurations:



Here cod is codomain. The remaining such events are directly promoted.

4 Discussion

Some of the benefits of diagrammatic notations are evident in the formal specification developed here where set intersection, disjointness and containment are represented visually. The diagrams presented here have properties that are thought to correlate with areas where diagrams are superior, from a usability perspective, to symbolic notations because they are well-matched to their set-theoretic semantics [9]. Extending this observation, using containment to represent set inclusion has the added benefit that the transitive property of the (semantic) subset relation is mirrored by the transitive property of (syntactic) containment. Any notation that is based on Euler diagrams to make such statements about sets is well-matched to its semantics.

The economy of syntax afforded by diagrams over symbolic notations is also sometimes an advantage. In diagrammatic specification of the class *VCP*, the relative placement of the *Track*, *Root* and *Rev* curves gives, for free, that *Root* is disjoint from *Rev*. This example of a *free ride*, the theory of which is developed by Shimojima [18], is an instance of where the explicit information in a diagram includes facts that would need to be inferred in the symbolic case. Other types of free rides arise and are not solely an advantage of Euler diagrams; for example, see the discussions on various types of free rides in constraint diagrams that relate to their arrows [11]. This type of inferential advantage of diagrams has been noted by several researchers, including Barwise and Etchemendy [3] and Stenning and Lemon [24], and is backed up by empirical evidence provided by Shimojima and Katagiri [19]. The advantages of diagrams in numerous reasoning contexts are further discussed by Larkin and Simon [14].

Tool Support

Significant tool support has been developed for using symbolic notations for specification and reasoning. However, the visualizations available to the users are not as sophisticated as those possible with the notations proposed in this paper. It is possible to provide tool support for this diagrammatic framework. Key pieces of functionality include:

(i) The ability to input diagrams via an editor or sketch recognition system.

(ii) The ability to automatically translate diagrams into symbolic forms to enable us to take advantage of the significant tool support that has been developed to date, including highly efficient reasoners. Moreover, it is desirable to support the translation of symbolic statements into a diagrammatic form, permitting their visualization.

(iii) The provision of a proof assistant or automated theorem prover which can be used to allow users to explore the logical consequences of their diagrammatic specifications.

(iv) The ability to automatically generate diagrams, in particular to support automated reasoning and visualization of symbolic statements.

In the latter case above, significant research has been directed towards the automated generation and layout of Euler diagrams, which form the bases of constraint diagrams, including [4,6,25]. Theorem provers have been developed for Euler diagram [23] and spider diagrams [7]. There is already a firm basis on which we can build in order to further develop functional tools for diagrammatic specification.

5 Conclusion

This paper presented a visual framework for specification and refinement and used it to develop an abstract model for a transparent configuration control platform. The specification was built up by refinement; its successive levels reflect separate requirements of configuration control, with increasingly-complex constraints. Of course, this model can be refined much further. We have illustrated the ideas in this paper by using this case study. A formalization of this particular diagrammatic framework is being developed and builds on the formalization of similar notations such as spider diagrams [12] and constraint diagrams [5,22]. We plan to present explicitly the diagrammatic rules of refinement that were illustrated in this paper and to formalize them. A longer term aim is to produce a diagrammatic version of Back's refinement calculus [2].

The diagrammatic notation developed here can also be viewed as way a visualizing standard symbolic notations such as B [1]. We argue that diagrammatic formal specification and refinement can be easier for clients to understand and there can, therefore, be benefits when using them.

Acknowledgement

This research was in part supported by the UK EPSRC grants EP/E011160 and EP/H012311, for the *Visualization with Euler Diagrams* and *Defining Regular Languages with Diagrams* projects, respectively.

References

[1] J. R. Abrial. *The B-Book: assigning programs to meanings.* CUP, 1996.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: a systematic introduction.* Springer, 1998.

[3] J. Barwise and J. Etchemendy. *Logical Reasoning with Diagrams*, Visual Information and Valid Reasoning. OUP, 1990.

[4] S. Chow and F. Ruskey. Drawing area-proportional Venn and Euler diagrams. In *Graph Drawingy*, pages 466–477. Springer-Verlag, 2003.

[5] Fish A, Flower J, Howse J (2005) The semantics of augmented constraint diagrams. *Journal of Visual Languages and Computing* **16**, 541–573

[6] J. Flower and J. Howse. Generating Euler diagrams. In *2nd International Conference on the Theory and Application of Diagrams*, pages 61–75, 2002. Springer.

[7] J. Flower, J. Masthoff, and G. Stapleton. Generating readable proofs: A heuristic approach to theorem proving with spider diagrams. In *3rd International Conference on the Theory and Application of Diagrams*, pages 166–181, 2004. Springer.

[8] Gil J, Howse J, Kent S (2001) Towards a formalization of constraint diagrams. In: *Proc. Symposium on Human Centric Computing 2001*, 72–79

[9] C. Gurr. Aligning syntax and semantics in formalisations of visual languages. In *IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 60–61. IEEE, 2001.

[10] Howse J, Schuman, S (2005) Precise visual modeling: a case-study. *Software and Systems Modeling Journal* **4**(3), 310–325

[11] J. Howse and G. Stapleton. Visual mathematics: Diagrammatic formalization and proof. In *International Conference on Mathematical Knowledge Management*, pages 478–493. Springer, 2008.

[12] Howse J., Stapleton G., Taylor J. Spider Diagrams. Available at The LMS Journal of Computation and Mathematics Vol.8 145-194, 2005.

[13] Kent S (1997) Constraint diagrams: visualising invariants in object oriented models. In: *Proc. OOPSLA97*, *SIGPLAN Notices* **32**(10), 327–341

[14] J. Larkin and H. Simon. Why a diagram is (sometimes) worth ten thousand words. *Journal of Cognitive Science*, 11:65–99, 1987.

[15] Pitt D, Byers P (1994) The rest stays unchanged (concurrency and state-based specification). *Formal Aspects of Computing* **6**, 471–494

[16] Schuman SA, Pitt DH (1987) Object-oriented subsystem specification. In: Meertens (ed.) *Program Specification and Transformation*, Proc. IFIP Working Conference, North-Holland, 313–341

[17] Schuman SA, Pitt DH, Byers PJ (1990) Object-oriented process specification. In: Rattray (ed.) *Specification and Verification of Concurrent Systems*, Proc. FACS Workshop, Springer, 21–70

[18] A. Shimojima. Inferential and expressive capacities of graphical representations: Survey and some generalizations. *Diagrams 2004*, pages 18–21. Springer, 2004.

[19] A. Shimojima and Y. Katagiri. An eye tracking study of spatial constraints in diagrammatic reasoning. *Diagrams 2008*, pages 64–88. Springer, 2008.

[20] Sommerville I, Sawyer P (1997) *Requirements engineering – a good practice guide*. Wiley

[21] G. Stapleton and A. Delaney. Evaluating and generalizing constraint diagrams. *Journal of Visual Languages and Computing*, 19(4):499–521, 2008.

[22] Stapleton G, Howse J, Taylor J A Decidable Constraint Diagram Reasoning System. In: *Journal of Logic and Computation* 15(6) 975-1008, 2005.

[23] G. Stapleton, J. Masthoff, J. Flower, A. Fish, and J. Southern. Automated theorem proving in Euler diagrams systems. *Journal of Automated Reasoning*, 39:431–470, 2007.

[24] K. Stenning and O. Lemon. Aligning logical and psychological perspectives on diagrammatic reasoning. *Artificial Intelligence Review*, 15(1-2):29–62, 2001.

[25] A. Verroust and M.-L. Viaud. Ensuring the drawability of Euler diagrams for up to eight sets. *Diagrams 2004*, pages 128–141. Springer, 2004.