WorldModel Template B/wmt

Purpose

Testing and changing the fundamental spatial relations.

§**1. The Core Tree.** Whereas I6 traditionally has a simple object tree hierarchy for containment, support, carrying and so on, the I7 template also uses the `component_*` properties to provide a second tree which defines the "part of" relation. These two trees interact in a subtle way, and it took a very long time to work out the simplest way to express this.

The *core* of an object is the root of its subtree in the component relation tree. So for a television set with a control panel which has a button on, the core for the button (and also for the panel and the set) will be the set. When X is a part of Y, it must be spatially in the same position as Y, and so the spatial location of X is determined by the position in the object tree of its core. (In effect, the spatial situation can be found by contracting together all nodes corresponding to objects which are parts of each other, but it would waste memory to construct such a tree: `CoreOfParentOfCoreOf` simulates what the `parent` operation would be in such a tree if it existed, wasting a little time instead.)

The *holder* of an object is its component parent, if it is part of something, or its object-tree parent if it has one. It is illegal for both of these to be non-`nothing`, so this is unambiguous.

It is just possible for `HolderOf` to be called before the player has been placed into the model world, in cases where Inform is checking a past tense condition in the opening pre-turn. We don't want to return `nothing` then because that would make it true that

```
HolderOf(player) == ContainerOf(player)
```

and similar conditions – thus, it would appear that in the immediate past the player had been on the holder of the player, which is not in fact the case. So we return `thedark` as a typesafe but impossible value here.

```
[ HolderOf o;
    if (InitialSituation-->DONE_INIS == false) return thedark;
    if (o && (o.component_parent)) return o.component_parent;
    if (o && (parent(o))) return parent(o);
    return nothing;
];
[ ParentOf o;
    if (o) o = parent(o);
    return o;
];
[ CoreOf o;
    while (o && (o provides component_parent) && (o.component_parent)) o = o.component_parent;
    return o;
];
[ CoreOfParentOfCoreOf o;
    while (o && (o provides component_parent) && (o.component_parent)) o = o.component_parent;
    if (o) o = parent(o);
    while (o && (o provides component_parent) && (o.component_parent)) o = o.component_parent;
    return o;
];
```

§**2. Climbing the Core Tree.** `LocationOf` returns the room in which an object can be found, or `nothing` if it is out of play. For this purpose a backdrop has no location, there being no canonical choice to make, while a two-sided door is in its "front side" (see below). Directions and regions are always out of play. For a room, `LocationOf` is necessarily itself.

`CommonAncestor` finds the nearest object indirectly containing `o1` and `o2`, or returns `nothing` if there is no common ancestor. (This is a port of `CommonAncestor` from the I6 library, adapted to work on the core tree.)

```
[ LocationOf o;
    if (~~(O ofclass K1_room or K2_thing)) return nothing;
    if (O ofclass K4_door) return FrontSideOfDoor(O);
    if (O ofclass K7_backdrop) return nothing;
    while (o) {
        if (o ofclass K1_room) return o;
        o = CoreOfParentOfCoreOf(o);
    }
    return nothing;
];
[ CommonAncestor o1 o2 i j;
    o1 = CoreOf(o1);
    o2 = CoreOf(o2);
    for (i=o1: i: i = CoreOfParentOfCoreOf(i))
        for (j=o2: j: j = CoreOfParentOfCoreOf(j))
            if (j == i) return j;
    return nothing;
];
[ IndirectlyContains o1 o2;
    if ((o1 == nothing) || (o2 == nothing)) rfalse;
    if ((o1 ofclass K1_room) && (o2 ofclass K4_door)) {
        if (o1 == FrontSideOfDoor(o2)) rtrue;
        if (o1 == BackSideOfDoor(o2)) rtrue;
        rfalse;
    }
    if (o2 ofclass K7_backdrop) rfalse;
    for (o2 = HolderOf(o2) : o2 && o2 ~= thedark : o2 = HolderOf(o2)) if (o2 == o1) rtrue;
    rfalse;
];
```

§**3. To Decide Whether In.** A curiosity, this: the I6 definition of "To decide whether in (obj - object)". As can be seen, there are three possible interpretations of "in", depending on the kind of object, so this could all be done with three different definitions in the Standard Rules, so that run-time type checking would decide which to apply: but that would produce a little overhead and make the code for this rather common operation a bit complicated. Besides, this way we can produce a respectable run-time problem message when the phrase is misapplied.

Note that "in X" is not equivalent to "the player is in X": the latter uses direct containment, whereas we use indirect containment. Thus "in the Hall" and "in the laundry-basket" are both true if the player is in a laundry-basket in the Hall.

```
[ WhetherIn obj;
    if (obj has enterable) {
        if (IndirectlyContains(obj, player)) rtrue;
        rfalse;
    }
    if (obj ofclass K9_region) return TestRegionalContainment(real_location, obj);
    if (obj ofclass K1_room) {
        if (obj == real_location) rtrue;
        rfalse;
    }
    RunTimeProblem(RTP_NOTINAROOM, obj);
    rfalse;
];
```

§**4. Containment Relation.** This is the single most important relation in I7: direct containment. It is complicated by the fact that "A is in B" is represented differently at run-time when A is a room and B is a region, and when it isn't.

For each A there is at most one B such that "A is in B" is true. Because of this we test the relation with a function of one variable which turns A into B, rather than a function of two variables returning true or false. `ContainerOf` is that function.

I7 frequently needs to compile loops over all A such that "A is in B". In simpler relations (such as the support relation below) it can do this efficiently by iterating through the object-tree children of B, but for containment we have to provide an iterator function `TestContainmentRange`, as otherwise we would get the wrong result when B is a region.

```
[ ContainerOf A p;
    if (A ofclass K1_room) return A.map_region;
    p = parent(A);
    if (p == nothing) return nothing;
    if (p ofclass K5_container) return p;
    if (p ofclass K1_room) return p;
    if (p ofclass K9_region) return p;
    return nothing;
];
[ TestContainmentRange obj e f;
    if (obj ofclass K9_region) {
        objectloop (f ofclass K1_room && f.map_region == obj)
            if (f > e) return f;
        return nothing;
    }
    if (obj ofclass K5_container or K1_room) {
        if (e == nothing) return child(obj);
```

```
        return sibling(e);
    }
    return nothing;
];
```

§**5. Support Relation.** This then is simpler, with no need for an iterator governing searches.

```
[ SupporterOf obj p;
    p = parent(obj);
    if (p == nothing) return nothing;
    if (p ofclass K6_supporter) return p;
    return nothing;
];
```

§**6. Carrying Relation.** Only people may carry, and something worn is not in this sense carried.

```
[ CarrierOf obj p;
    p = parent(obj);
    if (p && (p ofclass K8_person) && (obj hasnt worn)) return p;
    return nothing;
];
```

§**7. Wearing Relation.** Only people may wear.

```
[ WearerOf obj p;
    p = parent(obj);
    if (p && (p ofclass K8_person) && (obj has worn)) return p;
    return nothing;
];
```

§**8. Having Relation.** A person has something if and only if he either wears or carries it.

```
[ OwnerOf obj p;
    p = parent(obj);
    if (p && (p ofclass K8_person)) return p;
    return nothing;
];
```

§**9. Making Parts.** Note that `MakePart` removes the part-to-be from the object tree before attaching it: it cannot, of course, be the core of the resulting object.

```
[ MakePart P Of First;
    if (parent(P)) remove P; give P ~worn;
    if (Of == nothing) { DetachPart(P); return; }
    if (P.component_parent) DetachPart(P);
    P.component_parent = Of;
    First = Of.component_child;
    Of.component_child = P; P.component_sibling = First;
];
[ DetachPart P From Daddy O;
    Daddy = P.component_parent; P.component_parent = nothing;
    if (Daddy == nothing) { P.component_sibling = nothing; return; }
    if (Daddy.component_child == P) {
        Daddy.component_child = P.component_sibling;
        P.component_sibling = nothing; return;
    }
    for (O = Daddy.component_child: O: O = O.component_sibling)
        if (O.component_sibling == P) {
            O.component_sibling = P.component_sibling;
            P.component_sibling = nothing; return;
        }
];
```

§**10. Movements.** Note that an object is detached from its component parent, if it has one, when moved.

```
[ MoveObject F T opt going_mode was;
    if (F == nothing) return RunTimeProblem(RTP_CANTMOVENOTHING);
    if (F ofclass K7_backdrop) {
        if (T ofclass K9_region) {
            give F ~absent; F.found_in = T.regional_found_in;
            if (TestRegionalContainment(LocationOf(player), T)) move F to LocationOf(player);
            else remove F;
            return; }
        return RunTimeProblem(RTP_BACKDROP, F, T);
    }
    if (~~(F ofclass K2_thing)) return RunTimeProblem(RTP_NOTTHING, F, T);
    if (T ofclass K9_region) return RunTimeProblem(RTP_NOTBACKDROP, F, T);
    if (F has worn) {
        give F ~worn;
        if (F in T) return;
    }
    DetachPart(F);
    if (going_mode == false) {
        if (F == player) { PlayerTo(T, opt); return; }
        if ((IndirectlyContains(F, player)) && (LocationOf(player) ~= LocationOf(T))) {
            was = parent(player);
            move player to real_location;
            move F to T;
            PlayerTo(was, true);
            return;
```

```
        }
    }
    move F to T;
];
[ RemoveFromPlay F;
    if (F == nothing) return RunTimeProblem(RTP_CANTREMOVENOTHING);
    if (F == player) return RunTimeProblem(RTP_CANTREMOVEPLAYER);
    if (F ofclass K4_door) return RunTimeProblem(RTP_CANTREMOVEDOORS);
    give F ~worn; DetachPart(F);
    if (F ofclass K7_backdrop) give F absent;
    remove F;
];
```

§**11. On Stage.** The following implements the "on-stage" and "off-stage" adjectives provided by the Standard Rules. Here, as above, note that the I6 attribute `absent` marks a floating object (see below) which has been removed from play; in I7 only doors and backdrops are allowed to float, and only backdrops are allowed to be removed from play.

```
[ OnStage O set x;
    if (O ofclass K1_room) rfalse;
    if (set < 0) {
        while (metaclass(O) == Object) {
            if (O ofclass K1_room) rtrue;
            if (O ofclass K9_region) rfalse;
            if (O ofclass K4_door) rtrue;
            if (O ofclass K7_backdrop) { if (O has absent) rfalse; rtrue; }
            x = O.component_parent; if (x) { O = x; continue; }
            x = parent(O); if (x) { O = x; continue; }
            rfalse;
        }
    }
    x = OnStage(O, -1);
    if ((x) && (set == false)) RemoveFromPlay(O);
    if ((x == false) && (set)) MoveObject(O, real_location);
    rfalse;
];
```

§**12. Moving the Player.** Note that the player object can only be moved by this routine: this allows us to maintain the invariant for `real_location` and `location` (for which, see "Light.i6t") and to ensure that multiply-present objects can be witnessed where they need to be.

```
[ PlayerTo newplace flag;
    @push actor; actor = player;
    move player to newplace;
    location = LocationOf(newplace);
    real_location = location;
    MoveFloatingObjects();
    SilentlyConsiderLight();
    DivideParagraphPoint();
    if (flag == 0) <Look>;
    if (flag == 1) give location visited;
    if (flag == 2) AbbreviatedRoomDescription();
    @pull actor;
];
```

§**13. Move During Going.** The following routine preserves the invariant for `real_location`, but gets `location` wrong since it doesn't adjust for light. Nor are floating objects moved. It should be used only in the course of other operations which get the rest of this right. (I7 uses it only for the "going" action, where these various operations are each handled by different named rules to increase the flexibility of the system.)

```
[ MoveDuringGoing F T;
    MoveObject(F, T, 0, true);
    if (actor == player) {
        location = LocationOf(player);
        real_location = location;
    }
];
```

§**14. Being Everywhere.** The following is used as the `found_in` property for any backdrop which is "everywhere", that is, which is found in every room.

```
[ FoundEverywhere; rtrue; ];
```

§**15. Changing the Player.** It is very important that nobody simply change the value of the `player` variable, because so much else must be updated when the identity of the player changes: the light situation, floating objects, the `real_location` and so on. Because of this, the NI compiler contains code which compiles an assertion of the proposition "is(`player`, *X*)" into a function call `ChangePlayer(X)` rather than a variable assignment `player = X`. So we cannot catch out the system by writing "now the player is Mr Henderson".

We must ensure that if `player` is initially X, is changed to Y, and is then changed back to X, that both X and Y end exactly as they began – hence the flummery below with using `remove_proper` to ensure that `proper` is left with the correct value. Note that:

(1) at any given time exactly one person has the I6 `concealed` attribute: the current player;
(2) the `selfobj` is the default initial value of `player`, and because it has as its actual printed name "yourself", we need to override this when something else takes over as player: we change to "your former self", in fact. No such device is needed for other people being changed from because they are explicitly given printed names – say "Mr Darcy", "the sous-chef", etc. – in the source text.

```
[ ChangePlayer obj flag i;
    if (~~(obj ofclass K8_person)) return RunTimeProblem(RTP_CANTCHANGE, obj);
    if (~~(OnStage(obj, -1))) return RunTimeProblem(RTP_CANTCHANGEOFFSTAGE, obj);
    if (obj == player) return;
    give player ~concealed;
    if (player has remove_proper) give player ~proper;
    if (player == selfobj) {
        player.saved_short_name = player.short_name; player.short_name = FORMER__TX;
    }
    player = obj;
    if (player == selfobj) {
        player.short_name = player.saved_short_name;
    }
    if (player hasnt proper) give player remove_proper; ! when changing out again
    give player concealed proper;
    location = LocationOf(player); real_location = location;
    MoveFloatingObjects();
    SilentlyConsiderLight();
];
```

§**16. Floating Objects.** A single object can only be in one position in the object tree, yet backdrops and doors must be present in multiple rooms. This is accomplished by making them, in I6 jargon, "floating objects": objects which move in the tree whenever the player does, so that – from the player's perspective – they are always present when they should be. In I6, almost anything can be made a floating object, but in I7 this is strictly and only used for backdrops and two-sided doors.

There are several conceptual problems with this scheme: chiefly that it assumes that the only witness to the spatial arrangement of objects is the player. In I6 that was usually true, but in I7, where every person can undertake actions, it really isn't true any longer: if the objects float to follow the player, it means that they are not present with other people who might need to interact with them. This is why the accessibility rules are somewhat hacked for backdrops and doors (see "Light.i6t"). In fact we generally achieve the illusion we want, but this is largely because it is difficult to catch out all the exceptions for backdrops and doors, and because in practice authors tend not to set things up so that the presence or absence of backdrops much affects what non-player characters do.

All the same, the scheme is not logically defensible, and this is why we do not allow the user to create new categories of floating objects in I7.

The I6 implementation of `MoveFloatingObjects` acted on the `location` rather than the `real_location`, which (a) meant that multiply-present objects could include `thedark` – the generic Darkness place – as one possible

location, but (b) assumed that the only sense by which the player could witness an object was sight. In I7, `thedark` is not a valid room, and we are a bit more careful about the senses.

```
[ MoveFloatingObjects i k l m address flag;
    if (real_location == nothing) return;
    objectloop (i) {
        address = i.&found_in;
        if (address ~= 0 && i hasnt absent) {
            if (ZRegion(address-->0) == 2) {
                m = address-->0;
                .TestPropositionally;
                if (m.call(real_location) ~= 0) move i to real_location;
                else remove i;
            }
            else {
                k = i.#found_in;
                for (l=0 : l<k/WORDSIZE : l++) {
                    m = address-->l;
                    if (ZRegion(m) == 2) jump TestPropositionally;
                    if (m == real_location || m in real_location) {
                        if (i notin real_location) move i to real_location;
                        flag = true;
                    }
                }
                if (flag == false) { if (parent(i)) remove i; }
            }
        }
    }
];
[ MoveBackdrop bd D x address;
    if (~~(bd ofclass K7_backdrop)) return RunTimeProblem(RTP_BACKDROPONLY, bd);
    if (bd.#found_in > WORDSIZE) {
        address = bd.&found_in;
        address-->0 = D;
    } else bd.found_in = D;
    give bd ~absent;
    MoveFloatingObjects();
];
```

§**17. Wearing Clothes.** An object X is worn by a person P if and only if (i) the object tree `parent` of X is P, and (ii) X has the `worn` attribute. In fact for I7 purposes we are careful to ensure that (ii) happens only when (i) does in any case: if X is moved in the object tree, or made a part of something, or removed from play, then `worn` is removed.

```
[ WearObject X P opt;
    if (X == false) rfalse;
    if (X notin P) MoveObject(X, P, opt);
    give X worn;
];
```

§**18. Map Connections.** `MapConnection` returns the room which is in the given direction (as an object) from the given room: it is used, among other things, to test the "mapped east of" and similar relations. (The same relations are asserted true with `AssertMapConnection` and false with `AssertMapUnconnection`.)

`RoomOrDoorFrom` returns either the room or door which is in the given direction, and is thus simpler (since it doesn't have to investigate what is through such a door).

Both routines return values which are type-safe in I7 provided the kind of value they are assigned to is "object": neither returns any specific kind of object without fail. (`MapConnection` is always either a door or `nothing`, but `nothing` is not a typesafe value for "door".)

Note that map connections via doors are immutable.

```
[ MapConnection from_room dir
    in_direction through_door;
    if ((from_room ofclass K1_room) && (dir ofclass K3_direction)) {
        in_direction = Map_Storage-->
            ((from_room.IK1_Count)*No_Directions + dir.IK3_Count);
        if (in_direction ofclass K1_room) return in_direction;
        if (in_direction ofclass K4_door) {
            @push location;
            location = from_room;
            through_door = in_direction.door_to();
            @pull location;
            if (through_door ofclass K1_room) return through_door;
        }
    }
    return nothing;
];
[ DoorFrom obj dir rv;
    rv = RoomOrDoorFrom(obj, dir);
    if (rv ofclass K4_door) return rv;
    return nothing;
];
[ RoomOrDoorFrom obj dir use_doors in_direction sl through_door;
    if ((obj ofclass K1_room) && (dir ofclass K3_direction)) {
        in_direction = Map_Storage-->
            ((obj.IK1_Count)*No_Directions + dir.IK3_Count);
        if (in_direction ofclass K1_room or K4_door) return in_direction;
    }
    return nothing;
];
[ AssertMapConnection r1 dir r2 in_direction;
    SignalMapChange();
    in_direction = Map_Storage-->
        ((r1.IK1_Count)*No_Directions + dir.IK3_Count);
    if ((in_direction == 0) || (in_direction ofclass K1_room)) {
        Map_Storage-->((r1.IK1_Count)*No_Directions + dir.IK3_Count) = r2;
        return;
    }
    if (in_direction ofclass K4_door) {
        RunTimeProblem(RTP_EXITDOOR, r1, dir);
        return;
    }
    RunTimeProblem(RTP_NOEXIT, r1, dir);
```

```
];
[ AssertMapUnconnection r1 dir r2 in_direction;
    SignalMapChange();
    in_direction = Map_Storage-->
        ((r1.IK1_Count)*No_Directions + dir.IK3_Count);
    if (r1 ofclass K4_door) {
        RunTimeProblem(RTP_EXITDOOR, r1, dir);
        return;
    }
    if (in_direction == r2)
        Map_Storage-->((r1.IK1_Count)*No_Directions + dir.IK3_Count) = 0;
    return;
];
```

§**19. Adjacency Relation.** A relation between two rooms which, note, does not see connections through doors.

```
[ TestAdjacency R1 R2 i row;
    row = (R1.IK1_Count)*No_Directions;
    for (i=0: i<No_Directions: i++, row++)
        if (Map_Storage-->row == R2) rtrue;
    rfalse;
];
```

§**20. Regional Containment Relation.** This tests whether an object with a definite physical location is in a given region. (For a two-sided door which straddles regions, the front side is what counts; for a backdrop, a direction or another region, the answer is always no.) We rely on the fact that every room object has a `map_region` property which is the smallest region containing it, if any does, and `nothing` otherwise; region objects are then in the object tree such that parenthood corresponds to spatial containment. (Note that in I7, a region must either completely contain another region, or else have no overlap with it.)

```
[ TestRegionalContainment obj region o;
    if ((obj == nothing) || (region == nothing)) rfalse;
    if (~~(obj ofclass K1_room)) obj = LocationOf(obj);
    if (obj == nothing) rfalse;
    o = obj.map_region;
    while (o) {
        if (o == region) rtrue;
        o = parent(o);
    }
    rfalse;
];
```

§**21. Doors.** There are two sorts of door: one-sided and two-sided.

A two-sided door is in two rooms at once: one is called the front side, the other the back side. The front side is the one declared first in the source. Note that a one-sided door is also an I6 object of class `K4_door`, and then the front side is the room holding it, while the back side is `nothing`.

The `door_to` property calculates the room which the door leads to; that of course depends on which side of it the player is standing. Similarly, `door_dir` calculates the direction it leads in. Unlike the I6 setup, where `door_dir` returned the direction property (`n_to`, `s_to`, etc.), here in I7's template it returns the direction object (`n_obj`, `s_obj`, etc.)

```
[ FrontSideOfDoor D; if (~~(D ofclass K4_door)) rfalse;
    if (D provides found_in) return (D.&found_in)-->0; ! Two-sided
    return parent(D); ! One-sided
];
[ BackSideOfDoor D; if (~~(D ofclass K4_door)) rfalse;
    if (D provides found_in) return (D.&found_in)-->1; ! Two-sided
    return nothing; ! One-sided
];
[ OtherSideOfDoor D from_room rv;
    if (D ofclass K4_door) {
        @push location;
        location = LocationOf(from_room);
        rv = D.door_to();
        @pull location;
    }
    return rv;
];
[ DirectionDoorLeadsIn D from_room rv dir;
    if (D ofclass K4_door) {
        @push location;
        location = LocationOf(from_room);
        rv = D.door_dir();
        @pull location;
    }
    return rv;
];
```

§**22. Visibility Relation.** We use `TestScope` to decide whether there is a line of sight from A to B; it's a relation which cannot be asserted true or false.

```
[ TestVisibility A B;
    if (~~OffersLight(parent(CoreOf(A)))) rfalse;
    if (suppress_scope_loops) rtrue;
    return TestScope(B, A);
];
```

§**23. Touchability Relation.** We use `ObjectIsUntouchable` to decide whether there is physical access from A to B; it's a relation which cannot be asserted true or false.

```
[ TestTouchability A B;
    if (TestScope(B,A) == false) rfalse;
    if (ObjectIsUntouchable(B, 1, 0, A)) rfalse;
    rtrue;
];
```

§**24. Concealment Relation.** An activity determines whether one object conceals another; it's a relation which cannot be asserted true or false.

```
[ TestConcealment A B;
    if (A ofclass K2_thing && B ofclass K2_thing) {
        particular_possession = B;
        if (CarryOutActivity(DECIDING_CONCEALED_POSSESS_ACT, A)) rtrue;
    }
    rfalse;
];
```