Unifying Theories of Locations

Michael Anthony Smith[1,2] and Jeremy Gibbons[1]

[1] Oxford University, UK.
[2] Systems Assurance Group, QinetiQ, UK.
Michael.Smith@kellogg.ox.ac.uk
Jeremy.Gibbons@comlab.ox.ac.uk

Abstract. We present a Unifying Theories of Programming (UTP) model of locations, where a location is either *shareable* or *containable* depending on whether its value can be dereferenced by a pointer. Our model of locations is similar to previous work on pointers within the UTP; the main difference is that the previous work on pointers only modelled shareable locations. We explain why containable locations (whose values must be copied rather than aliased) are useful, present an outline of our UTP model, and compare it to existing work on UTP. We hope to convince the reader that a general model of pointers within the UTP ought to be able to represent both shareable and containable locations.

1 Introduction

Hoare and He's Unifying Theories of Programming (UTP) [3] uses the notion of a relational predicate to model various programming paradigms and features, such as imperative, functional, and parallel programming. Here, a relational predicate is a predicate that defines a relationship between observable input and output variables (i.e. the variables in the predicate's alphabet). For example, the UTP model in [1] supports the notion of a compound data structure via the introduction of a record datatype, which essentially maps distinct labels to values. These labels are also used when unambiguously specifying the location of a value and determining whether it is shared.

An object can be modelled in a similar manner to that of the record. For example, in C^{++} and C$^{\#}$ the object and record types are defined by the `class` and `struct` datatype constructors respectively. Here, a variable of an object type contains a pointer to an object, whereas a variable of a record type contains the record itself. It is this distinction between variables of object and record types that we believe is important to explicitly model in a general theory of UTP pointers. Specifically, the contents of a record are duplicated, whereas the contents of an object are aliased (shared).

The UML class diagram in Figure 1 provides a high level overview of our model, which ensures that: each location has precisely one value; only shareable locations can be directly accessed via a reference value; and field names (labels) of a compound value represent containable locations. Such a model of locations can be used to support our earlier UTP model of objects [9].

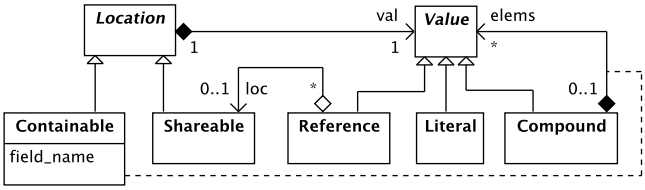


Fig. 1. Location Model – Class Diagram

1.1 Scope

The model of locations we present in this paper is not intended to support the concepts of object ownership or reference containment, such as discussed in ownership models [2, 5] and separation logics [7]. Nor is this model of locations intended to support low-level pointer operations, such as those operations that create a new pointer by adding an arbitrary offset to an existing pointer (e.g. $p = p+2$) or get the address of a record's element (e.g. $p = \&(r.x)$). Having said this, it is straightforward to write C++ and C# programs that do not directly use such low-level pointer operations and this ought to be syntactically checkable. For example, in C# this could be achieved by banning the use of the `unsafe` keyword.

1.2 Family tree

Within this paper we use instances of the family tree class diagram in Figure 2 to provide data structures for us to model. Here, shareable (hollow diamond) and composite (solid diamond) aggregations are used to distinguish between shareable and containable locations, respectively; aggregations that have any number of instances are represented by lists.

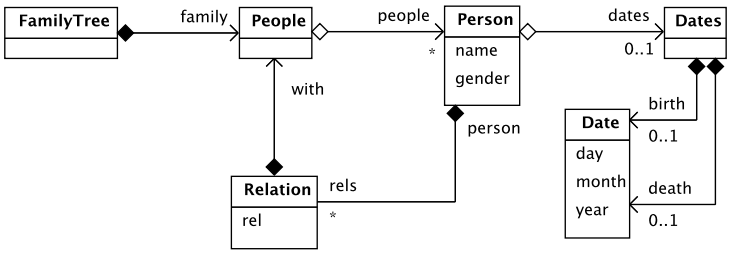


Fig. 2. Family Tree Example – Class Diagram

1.3 Structure

This paper continues by presenting a concrete model of locations (Section 2), which is abstracted (Section 3) and then integrated into the UTP (Section 4). Having done this, the work is related to other UTP work on pointers (Section 5) and summarised (Section 6).

2 Concrete Representation

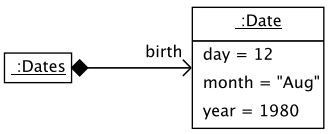
2.1 Concrete value notation

The two types of *literal value* used within this paper are the integers (e.g. -32) and the strings (e.g. "Some text"). There is also a special *unset* literal constant, denoted by ¿; this is used to represent the contents of a freshly created location, and the value of a missing element.

The two remaining types of concrete value are the *compound* and *reference values*. A compound value is represented by a partial map from field names (which we identify with containable locations) to concrete values. It is denoted by $\{^{k}_{i=1}\ nm_i = v_i\}$, where the name nm_i indexes the concrete value v_i. A name is denoted by an alpha-numeric word starting with a letter or the dollar symbol. The name represented purely by a single dollar symbol, which we refer to as the 'dollar name', is reserved for denoting a shareable location, and thus cannot be used as a compound value's field name.

A reference value is either null or an index to a shareable location. Such values are denoted by $\ominus$ and ℓ_i respectively, where two non-null reference values ℓ_i and ℓ_j index distinct shareable locations whenever $i \neq j$.

Figure 3 provides both an instance of the family tree's Dates class and a concrete value representation of this instance (object). Here, the Dates object explicitly sets only one of its two optional Date fields, birth, to 12 Aug 1980. The other optional field, death, is left unset. This data structure can be drawn as a



{ birth = { day = 12,
month = "Aug",
year = 1980 },
death = ¿
}

Fig. 3. Dates – object diagram and concrete representation

graph, as illustrated in Figure 4, where: a literal value is denoted by a boxed node containing the literal; and a compound value is denoted by a circular node, whose outgoing edges are labelled with its distinct field names. Reference values are denoted by a diamond node that contains the reference (Figures 6 & 8).

In addition to defining the concrete representations of the values used within this paper, it is useful to provide some meta-variables for representing each of

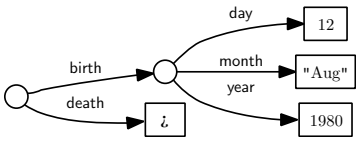


Fig. 4. Dates – concrete graph representation

the different types of value. Here we use i, j and k to represent integers; s to represent strings; lv to represent literal values; cv to represent compound values; rv to represent reference values; and v to represent a concrete value. We also use meta-variable t for representing a concrete term, where a concrete term includes the concrete values, field names, and the yet-to-be terms such as location graphs. These meta-variables are typically used to define functions, as illustrated by the following example, which extracts the shareable locations contained within a term. This example also uses the generalised term notation, $t\{\!\{_{i=1}^{k} t_i\}\!\}$, which denotes a term t with k sub-terms, $t_1, \ldots, t_k$, where a term that has no subterms is denoted by either $t\{\}$ or $t\{\!\{_{i=1}^{0} t_i\}\!\}$.

$$\begin{array}{lcl} sLocs\ \ell_i & \widehat{=} & \{\ell_i\} \\ sLocs\ t\{\!\{_{i=1}^{k} t_i\}\!\} & \widehat{=} & \bigcup\{\!\{_{i=1}^{k} sLocs\ t_i\}\!\} \end{array}$$

We read such definitions by pattern-matching from top to bottom, accepting the first equation that matches an actual argument. Thus, the order in which the lines of a function are presented may affect its meaning. In this case, swapping the order would produce a function that returns the empty set.

The *sLocs* function is applied to terms that are yet to be defined, such as the heap value term in Section 2.2. Note that this does not require an update to the *sLocs* function as these terms are already handled by the second definitional line, which can be applied to any term (i.e. a general term).

2.2 Concrete location heap

A *location heap* is a partial map from shareable locations to values; it is denoted by $\{_{i:N}\ \ell_i \mapsto v_i\}$, where N is some finite subset of the natural numbers. For example, the object diagram in Figure 5 illustrates that Jane Doe is married to John Doe, where only the instances of the Person class are considered to be shareable. It can be represented by the following concrete location heap, where

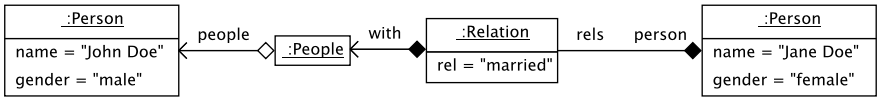


Fig. 5. Marriage example – object diagram

the contents of shared locations ℓ_3 and ℓ_5 contain the John Doe and Jane Doe Person objects respectively.

$$
\begin{array}{l}
\{\, \ell_3 \mapsto \{\, \mathsf{name} = \text{"John Doe"},\ \mathsf{gender} = \text{"male"},\ \mathsf{dates} = \ominus,\ \mathsf{rels} = ¿\,\},\\
\quad \ell_5 \mapsto \{\, \mathsf{name} = \text{"Jane Doe"},\ \mathsf{gender} = \text{"female"},\ \mathsf{dates} = \ominus,\\
\qquad\quad \mathsf{rels} = \{\, \$1 = \{\, \mathsf{rel} = \text{"married"},\ \mathsf{person} = \ell_5,\ \mathsf{with} = \{\, \mathsf{people} = \{\, \$1 = \ell_3 \,\}\,\}\,\}\,\}\\
\qquad \}\\
\}
\end{array}
$$

Figure 6 provides the alternative graph representation of the example, where the dashed edges are used to link a reference value to its contents. Note that these

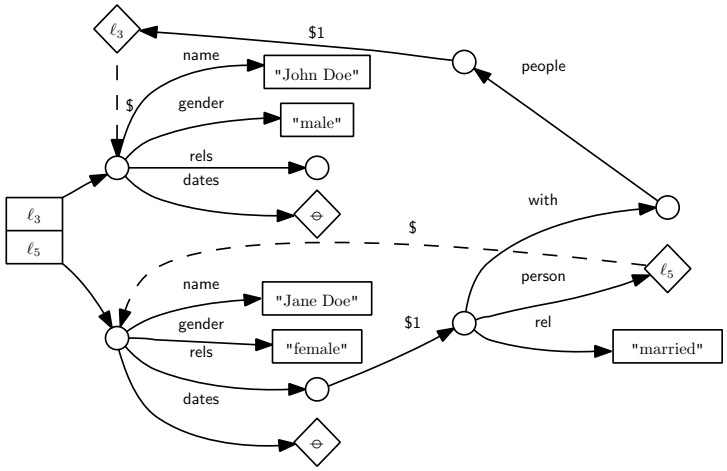


Fig. 6. Marriage example – concrete heap graph

edges are labelled with the dollar name ($) as discussed in Section 2.1.

Before moving on to present the concrete location model, we observe that a concrete heap can reference a shareable location that it does not define; i.e. a concrete heap can contain reference values that are not in the domain of the heap's partial map. In order to classify location heaps that do not have this undesirable property, we introduce a healthiness condition, which considers a heap to be healthy whenever all references to shared locations within the graph's values are defined by the graph itself.

$$\mathrm{HC}_{\mathrm{H}}\ H \ \widehat{=}\ \mathit{sLocs}\ H \subseteq \mathrm{dom}\ H$$

where:

H	is the meta variable representing a concrete location heap
$\mathrm{dom}\ \_$	returns the domain of a relation or function

Further, any heap can be made healthy by adding an entry for each missing

shared location and setting that location's value to the unset value (¿), as follows:

$$\mathrm{MH}_{\mathrm{H}}\ H \ \widehat{=}\ H \cup \{r \mapsto \text{¿} \mid r \in (sLocs\ H) \setminus (\mathrm{dom}\ H)\}$$

Note that a healthy heap is unaffected by the application of MH_{H} (and vice-versa), because all the shareable locations in the heap are contained within its domain; i.e. $(\mathrm{MH}_{\mathrm{H}}\ H = H) \Leftrightarrow \mathrm{HC}_{\mathrm{H}}\ H$.

The notion of equivalence ($\_ \equiv \_$) between location heaps is more complex than that for concrete values, which is mathematical equality (where the ordering of elements within a set or map is not significant). Here, the equivalence relationship between heaps allows for the renaming of shareable locations. Specifically, two heaps are considered to be equivalent if there exists a bijective map (f) that can be applied to one heap to produce the other.

$$H_1 \equiv H_2 \quad \Leftrightarrow \quad \exists f \bullet H_1 = rename(H_2, f)$$

where

$$\begin{array}{lcl} rename(\ell_i, f) & \widehat{=} & \ell_{f(i)} \\ rename(t\{\!|_{i=1}^{k} t_i|\!\}, f) & \widehat{=} & t\{\!|_{i=1}^{k} rename(t_i, f)|\!\} \end{array}$$

2.3 Concrete location model

Location models extend this notion of the heap by adding a starting point, which is represented by a concrete value. Therefore, a location model is denoted by a value-heap pair (v, H). Here, the idea is that the value v represents the root of a computational unit, such as a program, whose elements can share data via the shareable locations in the heap H.

Like the location heap that preceded it, location models have a healthiness condition which ensures that the heap is valid; that is, all the shareable locations referenced within a model are defined by the heap.

$$\mathrm{HC1}_{\mathrm{L}}(v, H) \ \widehat{=}\ sLocs(v, H) = \mathrm{dom}\ H$$

A location model can be made $\mathrm{HC1}_{\mathrm{L}}$-valid in a similar manner to a heap.

$$\mathrm{MH1}_{\mathrm{L}}(v, H) \ \widehat{=}\ (v, H \cup \{r \mapsto \text{¿} \mid r \in sLocs(v, H) \setminus (\mathrm{dom}\ H)\})$$

Locations in the model are considered to be *reachable* if they are either contained within the starting value v or indirectly contained within the contents of v's reference values. For $\mathrm{HC1}_{\mathrm{L}}$-healthy models, this can be formalised by the following functions, where R and R' represent the shareable locations that have already been taken into account and are contained within a value respectively.

$$\begin{array}{lcl} reachable(v, H) & \widehat{=} & reachValue(v, H, \emptyset) \\ reachValue(v, H, R) & \widehat{=} & sLocs\ v \cup reachDeref(sLocs\ v, H, R) \\ reachDeref(R', H, R) & \widehat{=} & \bigcup\{reachValue(H\ r, H, R' \cup R) \mid r \in R' \setminus R\} \end{array}$$

The following normal-form healthiness condition ensures that there is no unreachable information within the model; i.e. every shareable location that is defined by a model's heap is reachable.

$$\mathrm{HC2_L}(v, H) \;\widehat{=}\; \mathrm{dom}\, H = \mathit{reachable}(\mathrm{MH1_L}(v, H))$$

Note that we apply the $\mathrm{MH1_L}$ healthiness constructor prior to performing the reachability calculation, in order to ensure that $\mathrm{HC2_L}$ calculation is defined. If a location model is not in normal form (i.e. $\mathrm{HC2_L}$-healthy), it can be made so by ensuring that it is $\mathrm{HC1_L}$-healthy and then removing all the unreachable locations.

$$\begin{aligned}\mathrm{MH2_L}(v, H) \;\widehat{=}\; &\mathsf{let}\ (v_1, H_1) \mathrel{\widehat{=}} \mathrm{MH1_L}(v, H)\ \mathsf{in} \\ &(\, v_1, \{rv \mid rv \in H_1 \wedge (\mathit{first}\ rv) \in \mathit{reachable}(v_1, H_1)\}\,)\end{aligned}$$

where $\mathit{first}(x, y) \mathrel{\widehat{=}} x$.

We are now in a position to define an equivalence relation over location models. It is similar to that of heaps, except that we first ensure that models are made healthy before performing the check, as we only want to consider reachable elements in a model's heap. In other words, two location models are equivalent iff there exists some bijective shareable-location-renaming function f that enables two normalised heaps to be made equal.

$$\begin{aligned}&(v_1, H_1) \equiv (v_2, H_2) \\ &\Leftrightarrow \\ &\exists f \bullet \mathrm{MH2_L}(v_1, H_1) = \mathit{rename}(\mathrm{MH2_L}(v_2, H_2), f)\end{aligned}$$

It is this notion of equivalence up to which our UTP model of locations is fully abstract, as described in Section 4.3.

The family tree example can now be extended to illustrate a concrete location model, by adding an object to represent the family tree, as illustrated in Figure 7. The concrete graph representation of this example is provided by Figure 8, where

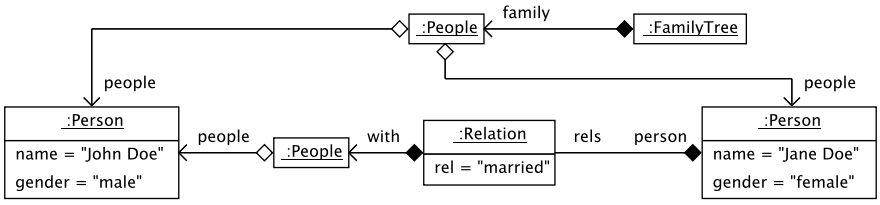


Fig. 7. Family tree example – object diagram

the explicit visualisation of the heap has been removed, as it is no longer required for representing the shareable locations. Such locations are now represented by the dashed edges within the graph, which are now guaranteed to exist due to the reachability healthiness condition.

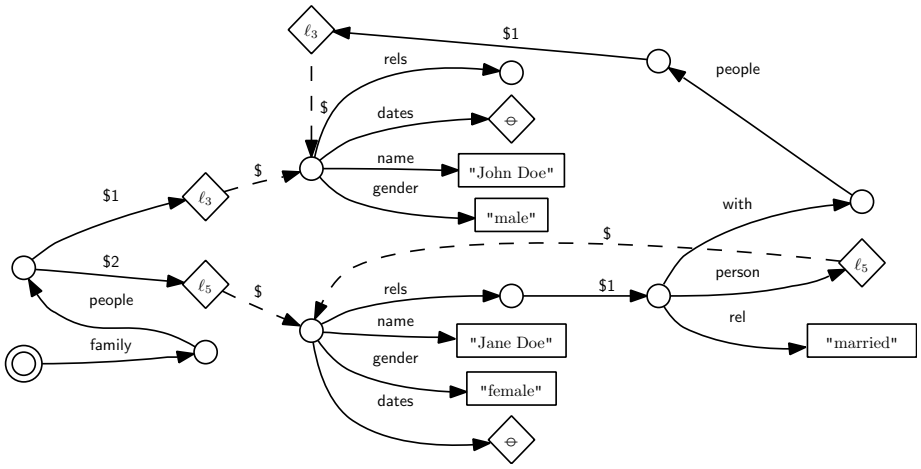


Fig. 8. Family tree example – concrete model graph

2.4 Paths and their operations

A *compound value path* describes a route from a compound value to one of its elements, via a non-empty dot-separated sequence of field names. Compound value paths are essentially used to describe routes to contained locations, which we can access and update by using the following functions:

$$
\begin{array}{lcl}
*cv.nm & \widehat{=} & cv\ \ nm \\
cv.nm := v & \widehat{=} & cv \oplus \{nm \mapsto v\} \\
cv.nm.p := v & \widehat{=} & cv \oplus \{nm \mapsto (*cv.nm).p := v\}
\end{array}
$$

where p is the meta-variable for paths, $(\_\ \_)$ is the function or map application operation, and $(\_ \oplus \_)$ is the function override operation. This notion of a path is extended to define location model update and access functions as follows:

$$
\begin{array}{lcl}
*(v, L) & \widehat{=} & v \\
*(v, L).p & \widehat{=} & *v.p \\
*(v, L).\ell_i & \widehat{=} & L\ \ell_i \\
*(v, L).\ell_i.p & \widehat{=} & (L\ \ell_i).p \\
(v, L) := v' & \widehat{=} & (v', L) \\
(v, L).p := v' & \widehat{=} & (v.p := v', L) \\
(v, L).\ell_i := v' & \widehat{=} & (v, L \oplus \{\ell_i \mapsto v'\}) \\
(v, L).\ell_i.p := v' & \widehat{=} & (v, L \oplus \{\ell_i \mapsto (L\ \ell_i).p := v'\})
\end{array}
$$

Further, it is possible to extend this notion to copy a value from one location to another, as follows:

$$(v, L).lp := (v, L).lp' \quad \widehat{=} \quad (v, L).lp := *(v, L).lp'$$

where: $\varnothing$ denotes the empty path and lp denotes either a path (p), a shareable location index (ℓ_i), or a shareable location index followed by a path ($\ell_i.p$); and $(v, L).\varnothing$ denotes (v, L).

It is also straightforward to define other operations, such as for deleting elements from compound values and the heap; we omit these constructions for reasons of space.

3 Abstract Model

In Section 2.3, graphs represented healthy concrete location models, where:

- the solid and dashed edges denote distinct compound and potentially shared shareable locations, respectively;
- the rectangular, circular, and diamond nodes denote literal, compound, and reference values, respectively.

This section presents: a brief overview of the trace-based graph abstraction; some utility operations for manipulating traces; a model of nodes as a set of traces; and an overview of the trace-based location graph model.

3.1 Graph abstraction

We can determine the value of dereferencing a reference node of a concrete location graph by following that node's outbound edge (as shown in Figure 8). That is, the shareable location index contained within a reference node is not required. Thus, this unused data can, and will, be ignored in our abstraction.

We observe that the outbound edges of each node within a healthy concrete location graph have distinct labels. Therefore, we can use a finite non-empty sequence of names to unambiguously define a path from a graph's root node to any other node. Such a path is from now on referred to as an *absolute path*.

The location of a node within a concrete location graph can be modelled by the set of all absolute paths to that node, which we from now on refer to as an *absolute path-set*. Hence, one way of providing a UTP model of locations would be as a partial map from such an absolute path-set to an appropriate abstraction of the data directly associated with its corresponding node. For example, the data associated with:

- a literal or null-reference node could be modelled by its concrete value;
- a compound node could be modelled by its set of outbound edge labels;
- a non-null reference node could be modelled by its outbound edge label.

Such a model of locations is similar to that presented in [1], which uses the idea of an entity group to model shared locations. Here each group contains the set (equivalence class) of fully qualified variables that share the same location.

Another approach is to change the notion of an absolute path-set, from representing the location of a node to representing both the location and contents

of a node. To avoid confusion, we refer to such paths as *traces*. Here, the idea is that the last value in a trace represents its content, and the front of the trace its location. In other words, a trace is a path p followed by a trace label l, where l represents either a name, a literal value, or the null-reference value; it is denoted by $p.l$. This is the basis of the UTP model of locations we present in this paper (Section 4). Such a model of locations is similar to that presented in [4], which uses trace-sets to model both locations and values. Here, the main difference is in our introduction of a containable location and its effect on assignment (which [4] refers to as 'pointer swing'). Specifically, within our model the contents of contained locations need to be duplicated, whereas the contents of shared locations are referenced.

3.2 Traces

As previously stated, a trace is denoted by $p.l$, where p is a path and l is a trace label (i.e. a name, a literal, a null-reference). One consequence of this is that it is only possible to concatenate two traces (denoted by $tr_1.tr_2$) when the last label in the first trace tr_1 is a name, as only names are allowed within a path.

The remainder of this section defines some utility operations on traces and trace-sets, that are used in the construction of our abstract model of locations. First we introduce two operations front and last for extracting the location and content components of a non-empty trace.

$$\mathsf{front}\, p.l \mathrel{\widehat{=}} p \qquad \mathsf{last}\, p.l \mathrel{\widehat{=}} l$$

The front operation can be used to generate the set of locations visited by a trace, as characterised by their paths, where each path within this set is considered to be a *prefix* of the original trace. Such a set of paths is referred to as the *proper prefixes* of the given trace. The function $\mathsf{prefixes}_{\mathrm{T}}$ defines the non-proper version of the prefix set.

$$\begin{array}{ll} \mathsf{prefixes}_{\mathrm{T}}\, \varnothing & \mathrel{\widehat{=}} \{\varnothing\} \\ \mathsf{prefixes}_{\mathrm{T}}\, tr & \mathrel{\widehat{=}} \{tr\} \cup \mathsf{prefixes}_{\mathrm{T}}(\mathsf{front}\, tr) \end{array}$$

The prefixes also provide a natural ordering over traces.

$$\begin{array}{lll} tr_1 <_{\mathrm{T}} tr_2 & \mathrel{\widehat{=}} & \mathsf{prefixes}_{\mathrm{T}}(tr_1) \subset \mathsf{prefixes}_{\mathrm{T}}(tr_2) \\ tr_1 \leq_{\mathrm{T}} tr_2 & \mathrel{\widehat{=}} & \mathsf{prefixes}_{\mathrm{T}}(tr_1) \subseteq \mathsf{prefixes}_{\mathrm{T}}(tr_2) \end{array}$$

Having defined an ordering over traces, it is now possible to use that ordering to define a subtraction operation. This is eventually used to define the relative paths between nodes in a set.

$$(\_ -_{\mathrm{T}} \_) \mathrel{\widehat{=}} \lambda\, tr_1, tr_2 \mid tr_2 \leq_{\mathrm{T}} tr_1 \bullet \mathsf{pick}\{tr \mid tr_1 = tr_2.tr\}$$

where the pick function picks the singleton element from a set (i.e. $\mathsf{pick}\{x\} \mathrel{\widehat{=}} x$).

Before leaving the trace utilities, we lift the definitions of the front, last, and $\mathsf{prefixes}_{\mathrm{T}}$ operations to trace-sets. The first two are lifted by applying their definitions to each non-empty trace with the set. The latter one is lifted to a trace-set (denoted by TR) by applying the prefixes operation to each trace within the set and merging the results.

$$\begin{array}{lcl} \mathsf{frontTraces}\ TR & \widehat{=} & \{\mathsf{front}\ tr \mid tr \in (TR \setminus \{\varnothing\})\} \\ \mathsf{lastLabels}\ TR & \widehat{=} & \{\mathsf{last}\ tr \mid tr \in (TR \setminus \{\varnothing\})\} \\ \mathsf{prefixes}_{\mathrm{N}}\ TR & \widehat{=} & \bigcup\{\mathsf{prefixes}_{\mathrm{T}}\ tr \mid tr \in TR\} \end{array}$$

3.3 Trace-based node

A trace-based graph node is modelled by a set of traces that satisfies two healthiness conditions. Both of these conditions follow from the observation that the only way a concrete location graph node may have more than one incoming edge, is if all these edges are labelled with the dollar name. Consequently, every trace to a node is guaranteed to end with the same label, except for the root node which has no label. This is modelled by the first healthiness condition, which states that all incoming edges to a node have the same label.

$$\mathrm{HC1}_{\mathrm{N}}(n) \ \widehat{=}\ \#\,\mathsf{lastLabels}(n) \leq 1$$

Another consequence of the observation is that a node may only have multiple parents if it is stored in a shareable location. This is modelled by the second healthiness condition, which states that the trace to any node that has more than one parent must end with the special shareable location label.

$$\mathrm{HC2}_{\mathrm{N}}(n) \ \widehat{=}\ \#\,\mathsf{lastLabels}(\mathsf{frontTraces}(n)) > 1 \Rightarrow \mathsf{lastLabels}(n) = \{\$\}$$

Any healthy node can be denoted by $P.l$, where each path in the path-set P is extended by the trace-label l to form the trace-set $\{p.l \mid p \in P\}$. The remainder of this section now presents some useful utility relations and operations on nodes.

Node relations: The *child-of* and *descendant-of* relations test whether one node is an immediate child of or a descendant of another node. These tests assume that the nodes come from a healthy graph, where all the routes to the parent are contained within the child.

$$\begin{array}{lcl} n_1\ \mathsf{childOf}\ n_2 & \widehat{=} & n_2 \in \mathsf{frontTraces}(n_1) \\ n_1\ \mathsf{descendantOf}\ n_2 & \widehat{=} & n_2 \in \mathsf{prefixes}_{\mathrm{N}}(\mathsf{frontTraces}(n_1)) \end{array}$$

In addition to knowing whether two nodes are related, it is sometimes useful to identify the relative traces from a parent to child node.

$$\mathsf{traces}_{\mathrm{N}}(n_1, n_2) \ \widehat{=}\ \{tr_2 -_{\mathrm{T}} tr_1 \mid tr_1 \in n_1 \wedge tr_2 \in n_2 \wedge tr_1 \leq_{\mathrm{T}} tr_2\}$$

Such trace-sets are used to determine whether two nodes are related via shareable or via containable locations. Here, two nodes are related by a shareable location if one of the traces within the trace-set includes the dollar label. Similarly they are related by a containable location if one of the traces within the trace-set does not contain the dollar label.

$$\begin{array}{lcl} n_1 \ \mathsf{shareDescOf}\ n_2 & \widehat{=} & n_1 \ \mathsf{descendantOf}\ n_2 \wedge \$ \in_{\mathrm{T}} \mathsf{traces}_{\mathrm{N}}(n_2, n_1) \\ n_1 \ \mathsf{containDescOf}\ n_2 & \widehat{=} & n_1 \ \mathsf{descendantOf}\ n_2 \wedge \$ \notin_{\mathrm{T}} \mathsf{traces}_{\mathrm{N}}(n_2, n_1) \end{array}$$

where $l \in_{\mathrm{T}} tr \mathrel{\widehat{=}} \exists\, p \bullet p.l \in \mathsf{prefixes}_{\mathrm{T}}\ tr$.

Note that the only way a node can be both a shareable and a containable descendant of another node, is if the nodes are both contained in the same cycle. In this case, all the containable descendants are also shareable descendants.

Node unlinking (deletion): Part of the assignment process involves the removal of previously held data. This is the purpose of the following unlinking operations, which remove all traces of either a node (n_1) or its children from the specified target node (n_2).

$$\begin{array}{lcl} \mathsf{unlink}_{\mathrm{N}}\ n_1\ \mathsf{from}\ n_2 & \widehat{=} & n_2 \setminus \{tr_2 \mid tr_1 \in n_1 \wedge tr_2 \in n_2 \wedge tr_1 \leq_{\mathrm{T}} tr_2\} \\ \mathsf{unlinkChildren}_{\mathrm{N}}\ n_1\ \mathsf{from}\ n_2 & \widehat{=} & n_2 \setminus \{tr_2 \mid tr_1 \in n_1 \wedge tr_2 \in n_2 \wedge tr_1 <_{\mathrm{T}} tr_2\} \end{array}$$

These operations can be lifted to the graph context by unlinking a given node from a node-set.

$$\begin{array}{lcl} \mathsf{unlink}_{\mathrm{G}}(N, n) & \widehat{=} & \{(\mathsf{unlink}_{\mathrm{N}}\ n\ \mathsf{from}\ n') \mid n' \in N\} \\ \mathsf{unlinkChildren}_{\mathrm{G}}(N, n) & \widehat{=} & \{(\mathsf{unlinkChildren}_{\mathrm{N}}\ n\ \mathsf{from}\ n') \mid n' \in N\} \end{array}$$

Node duplication (replacement): It is sometimes useful to construct a new node from a pair of existing nodes, a source node (n_2) and one of its descendants (n_3). Here the idea is to extract the traces between the source and descendant nodes, and then append them to a new source node (n_1), which is the target of the duplication.

$$\mathsf{replace}_{\mathrm{N}}\ n_1\ \mathsf{for}\ n_2\ \mathsf{in}\ n_3 \quad \widehat{=} \quad \{tr_1.tr \mid tr_1 \in n_1 \wedge tr \in \mathsf{traces}_{\mathrm{N}}(n_2, n_3)\}$$

Instead of replacing one parent for another, we may want to add a parent; for example, when copying a reference to a shareable location. This is essentially achieved by performing the replacement operation and merging in the original data.

$$\mathsf{add}_{\mathrm{N}}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ n_3 \quad \widehat{=} \quad (\mathsf{replace}_{\mathrm{N}}\ n_1\ \mathsf{for}\ n_2\ \mathsf{in}\ n_3) \cup n_3$$

These operations can then lifted so that they operate on node-sets, by reparenting each node in the set.

$$\begin{array}{lcl} \mathsf{replace}_{\mathrm{G}}\ n_1\ \mathsf{for}\ n_2\ \mathsf{in}\ N & \widehat{=} & \{(\mathsf{replace}_{\mathrm{N}}\ n_1\ \mathsf{for}\ n_2\ \mathsf{in}\ n) \mid n \in N\} \\ \mathsf{add}_{\mathrm{G}}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ N & \widehat{=} & \{(\mathsf{add}_{\mathrm{N}}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ n) \mid n \in N\} \end{array}$$

We use these operations to prepare a subgraph for being moved or copied to a new location.

3.4 Trace-based graph

A *trace-based graph* is a set of trace-based nodes that satisfies four healthiness conditions. The first healthiness condition states that each of the graph's nodes is healthy.

$$\mathrm{HC1}_{\mathrm{G}}(G) \mathrel{\widehat{=}} \forall\, n \mid n \in G \bullet \mathrm{HC1}_{\mathrm{N}}(n) \wedge \mathrm{HC2}_{\mathrm{N}}(n)$$

The second healthiness condition states that the nodes of a graph are disjoint. This ensures that an absolute trace can be used to identify a single node.

$$\mathrm{HC2}_{\mathrm{G}}(G) \mathrel{\widehat{=}} \forall\, n_1, n_2 \mid \{n_1, n_2\} \subseteq G \wedge n_1 \neq n_2 \bullet n_1 \cap n_2 = \emptyset$$

For a graph (G) that satisfies condition $\mathrm{HC2}_{\mathrm{G}}$, it is possible to define an operation for extracting the node (n) that has an absolute trace (p), so long as the trace is within the graph.

$$\mathsf{node}_{\mathrm{G}}(G, tr) \mathrel{\widehat{=}} \lambda\, G, tr \mid \mathrm{HC2}_{\mathrm{G}}(G) \wedge tr \Subset_{\mathrm{G}} G \bullet \mathsf{pick}(\{n \mid tr \in n \in G\})$$

where the $(\_ \Subset_{\mathrm{G}} \_)$ relation determines whether a trace is in the graph:

$$p \Subset_{\mathrm{G}} G \mathrel{\widehat{=}} p \in (\bigcup G)$$

The third healthiness condition states that each of a node's traces is consistently extended; i.e. if it is possible to take an edge with label l from node n_1 to node n_2, then the trace-set formed by appending the label l to each of n_1's traces is a subset of n_2's trace-set.

$$\begin{aligned}\mathrm{HC3}_{\mathrm{G}}(G) \mathrel{\widehat{=}}\ & \forall\, n_1, n_2, tr_1, tr_2, l \mid \\ & \quad \{n_1, n_2\} \subseteq G \wedge \{tr_1, tr_2\} \in n_1 \bullet \\ & \quad\quad tr_1.l \in n_2 \Rightarrow tr_2.l \in n_2\end{aligned}$$

The fourth healthiness condition states that the parents of a node are contained within the graph; in other words, the traces within a graph are prefix closed.

$$\mathrm{HC4}_{\mathrm{G}}(G) \mathrel{\widehat{=}} \forall\, tr, l \mid tr.l \Subset_{\mathrm{G}} G \bullet tr \Subset_{\mathrm{G}} G$$

The combination of the first three graph healthiness conditions defines what it means for the trace model to have a consistent, but not necessarily complete, set of nodes. Thus, these conditions should be satisfied by any healthy subgraph.

The remainder of this section provides operations for manipulating the contents of a location graph model, such as operations for: extracting a subgraph; extracting the value at a location; and assigning a value to a location.

Children and descendants subgraphs: Subgraphs can be formed by selecting only some of a graph's nodes. The childOf and descendantOf relations can be used to filter a graph to form children and descendants subgraphs respectively.

$$\begin{aligned}\mathsf{children} &\mathrel{\widehat{=}} \lambda\, G, n \mid n \in G \bullet \{n' \mid n' \in G \wedge n'\ \mathsf{childOf}\ n\} \\ \mathsf{descendants} &\mathrel{\widehat{=}} \lambda\, G, n \mid n \in G \bullet \{n' \mid n' \in G \wedge n'\ \mathsf{descendantOf}\ n\}\end{aligned}$$

Note that a node can be a descendant of itself if, and only if, there is a non-empty sequence of edges back to itself.

Dereferencing a location's value: A location is represented by a healthy node whose last label is either a field name or the dollar name. Such a node can be represented by a path-set, as each trace within this node may only contain names. The value of a location node is determined by recursively examining its children, or more specifically its child labels. There are three cases to consider.

1. There is a single null-reference or literal value (nlv) child label. In this case, the label value is returned as the location's value.
2. There is a single child label that contains the dollar name. In this case the path-set (reference value) that models the child node is returned.
3. There is a set of child-labels that contain field names. In this case a compound value is recursively constructed from its children.

$$
\begin{array}{lcl}
*_{\mathrm{G}}(G, p) & \widehat{=} & *_{\mathrm{G}}(G, \mathsf{node}_{\mathrm{G}}(G, p)) \\
*_{\mathrm{G}}(G, P) & \widehat{=} & *_{\mathrm{G}}(G, \mathsf{children}(G, P)) \\
*_{\mathrm{G}}(G, \{P.nlv\}) & \widehat{=} & nlv \\
*_{\mathrm{G}}(G, \{P.\$\}) & \widehat{=} & P.\$ \\
*_{\mathrm{G}}(G, \{\overset{k}{_{i=1}} P.nm_i\}) & \widehat{=} & \{\overset{k}{_{i=1}} nm_i = *_{\mathrm{G}}(G, P.nm_i)\}
\end{array}
$$

Recall that we introduced $P.l$ as an alternative notation for denoting a healthy node, in Section 3.3, where $P.l \mathrel{\widehat{=}} \{p.l \mid p \in P\}$.

Preparing a location for assignment: The preparation required for assigning a value to a location depends on a number of factors, such as whether the location already exists. We could limit assignments to existing locations, but then this would not mirror our concrete model, which defined assignment in terms of the map overriding operation ($\_ \oplus \_$). Instead we categorise a potential location as either existing (E_{pm}), *freshly containable* (C_{pm}), *freshly shareable* (S_{pm}), or *invalid* (U_{pm}), as follows:

$$
prepMode(G, P.nm) \quad \widehat{=} \quad \begin{cases} E_{pm}, \text{ if } P.nm \in G \\ S_{pm}, \text{ if } P.nm \notin G \wedge nm = \$ \\ C_{pm}, \text{ if } P.nm \notin G \wedge nm \neq \$ \wedge *_{\mathrm{G}}(G, P) \in CV \\ U_{pm}, \text{ otherwise} \end{cases}
$$

where CV denotes the set compound values (i.e. the compound value type). Note that the above definition of freshly created locations ensures that a compound value may only contain containable locations (and vice versa). In general, a path-set P is considered to represent an assignable location within a graph G whenever it has a valid assignable location mode.

$$
P \ \mathsf{assignableIn}_{\mathrm{G}} \ G \quad \widehat{=} \quad prepMode(G, P) \neq U_{pm}
$$

It is now possible to define the preparation for an assignable location by ensuring that it exists and contains no contents. This can involve the clearing (unlinking)

of an existing node's contents and the creation of a new location node.

$$\begin{array}{lcl} \mathsf{prep}_{\mathrm{G}}(G, P) & \widehat{=\!=} & \mathit{prep}(G, P, \mathit{prepMode}(G, P)) \\ \mathit{prep}(G, P, E_{pm}) & \widehat{=\!=} & \mathsf{unlinkChildren}_{\mathrm{G}}(G, P) \\ \mathit{prep}(G, P.l, S_{pm}) & \widehat{=\!=} & \mathsf{unlinkChildren}_{\mathrm{G}}(G, P) \cup \{P.\$\} \\ \mathit{prep}(G, P.l, C_{pm}) & \widehat{=\!=} & G \cup \{P.l\} \end{array}$$

Assigning a null-reference or literal value: A null-reference or literal value (*nlv*) can be assigned to a graph location by preparing the location and setting its contents to the given value.

$$\begin{array}{lcl} (G, p) :=_{\mathrm{G}} \mathit{nlv} & \widehat{=\!=} & (G, \mathsf{node}_{\mathrm{G}}(G, p)) :=_{\mathrm{G}} \mathit{nlv} \\ (G, P) :=_{\mathrm{G}} \mathit{nlv} & \widehat{=\!=} & \mathsf{prep}_{\mathrm{G}}(G, P) \cup \{P.\mathit{nlv}\} \end{array}$$

Assigning an encapsulated compound value: An encapsulated compound value is a concrete compound value that contains no shareable locations (i.e. a compound value in the set $\{cv \mid \mathit{sLocs}\ cv = \emptyset\}$). Such values are represented by the meta-variable *ecv*. It can be assigned to a location by preparing the location and setting its contents to the subtree that represents the compound value.

$$\begin{array}{lcl} (G, p) :=_{\mathrm{G}} \mathit{ecv} & \widehat{=\!=} & (G, \mathsf{node}_{\mathrm{G}}(G, p)) :=_{\mathrm{G}} \mathit{ecv} \\ (G, P) :=_{\mathrm{G}} \mathit{ecv} & \widehat{=\!=} & \mathsf{prep}_{\mathrm{G}}(G, P) \cup \bigcup\{P.tr \mid tr \in (\mathit{cvTrs}\ \mathit{ecv})\} \end{array}$$

where the *cvTrs* function converts an encapsulated compound value into a prefix closed set of traces, representing each trace through the compound value's structure.

Assigning the contents of an existing location: In the concrete model, we referred to this as the copying of a location's value. This is more tricky than the previous cases for a number of reasons. One significant reason is that the location we are copying may be contained within the target location that we are assigning to. In such a case, the location preparation process could remove (clear) the location we want to copy. This limitation can be overcome by a three-step process. First, copy the value to a fresh temporary location, which is not contained within the contents of the target location. Second, prepare the target location and copy the value of the temporary location to it. Last, remove the temporary location.

What would make a good temporary location is dependant on what the location graph is being used to model, so in general we cannot specify this. Having said that, what we can do is specify how to assign the contents of a location to a prepared location node.

Assigning to a cleared location node: When assigning the contents of a cleared location, care has to be taken to ensure that the contents of reference values are pointed to rather than duplicated. In order to facilitate this, two utility operations are defined: one for identifying the referenced nodes (copyRefSG); and the

other to add the copied pointer (path) to these identified nodes (copyRefNodes).

$$\begin{array}{lcl} \mathsf{copyRefSG}(G, n) & \widehat{=} & \{n' \mid n' \in G \wedge n' \ \mathsf{shareDescOf}\ n\} \\ \mathsf{copyRefNodes}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ G & \widehat{=} & \mathsf{add}_{\mathrm{G}}\ n_2\ \mathsf{for}\ n_1\ \mathsf{in}\ \mathsf{copyRefSG}(G, n_1) \end{array}$$

Care also has to be taken to ensure that a duplicate of the value nodes are added to the copy node. This is facilitated by two utility operations: one for identifying the nodes to be duplicated (copyValSG); and the other to perform the duplication (copyValNodes) using the node replacement operation.

$$\begin{array}{lcl} \mathsf{copyValSG}(G, n) & \widehat{=} & \{n' \mid n' \in G \wedge n' \ \mathsf{containDescOf}\ n\} \\ \mathsf{copyValNodes}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ G & \widehat{=} & \mathsf{replace}_{\mathrm{G}}\ n_2\ \mathsf{for}\ n_1\ \mathsf{in}\ \mathsf{copyValSG}(G, n_1) \end{array}$$

It is now possible to define the graph transformation operation of copying the contents of a source node to the empty location as the union of: the appropriately updated reference nodes; the descendant nodes that were not updated; the non-descendant nodes; and the duplicated value nodes.

$$\begin{array}{lcl} \mathsf{copy}_{\mathrm{G}}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ G & \widehat{=} & (\mathsf{copyRefNodes}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ G) \\ & \cup & \mathsf{descendants}(G, n_1) \setminus \mathsf{copyRefSG}(G, n_1) \\ & \cup & G \setminus \mathsf{descendants}(G, n_1) \\ & \cup & (\mathsf{copyValNodes}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ G) \end{array}$$

Now given that the location with path $copy is an assignable location that does not exist within the graph, the copy assignment can be defined as follows:

$$\begin{array}{lcl} (G, p) :=_{\mathrm{G}} p' & \widehat{=} & (G, p) :=_{\mathrm{G}} \mathsf{node}_{\mathrm{G}}(G, p') \\ (G, p) :=_{\mathrm{G}} P' & \widehat{=} & (G, \mathsf{node}_{\mathrm{G}}(G, p)) :=_{\mathrm{G}} P' \\ (G, P) :=_{\mathrm{G}} P' & \widehat{=} & \mathsf{let}\ G_1 \mathrel{\widehat{=}} (\mathsf{copy}_{\mathrm{G}}\ P'\ \mathsf{to}\ \{\$\mathsf{copy}\}\ \mathsf{in}\ (G \cup \{\$\mathsf{copy}\})) \\ & & \quad\ G_2 \mathrel{\widehat{=}} (\mathsf{copy}_{\mathrm{G}}\{\$\mathsf{copy}\}\ \mathsf{to}\ P\ \mathsf{in}\ \mathsf{prep}_{\mathrm{G}}(G_1, P)) \\ & & \mathsf{in}\ \ \mathsf{unlink}_{\mathrm{G}}(G_2, \{\$\mathsf{copy}\}) \end{array}$$

4 UTP Model

Our UTP model of locations uses the Abstract Location Trace Graph (ALTG) of Section 3.4 to provide a semantics of locations, where the special logical variables *altg* and *altg′* to represent the before and after states of the graph. The contents of this ALTG are then linked to the normal UTP program variables, using a technique inspired by [1]. In our case, the values of normal program variables are mirrored by correspondingly named first-level nodes in the graph. For example, the logical input and output variables for a UTP program variable x are represented by the node $\{x\}$ in the *altg* and *altg′* graphs respectively. Note that whenever there could be confusion between whether a variable is being used to denote its name rather than its value, we prefix the variable with a dash to

get its name. For example, the predicate $x = *_{\mathrm{G}}(altg, \text{'}x)$ holds whenever the value of variable x equals the value of extracting its corresponding element from the graph *altg* (i.e. the one with the path name $\text{'}x$).

The remainder of this section introduces the healthiness conditions on the UTP model of locations, provides the definitions for a few operations, such as assignment, and relates the abstract and concrete models. Here the meta variable Q denotes a relational predicate that defines a UTP location model program.

4.1 Healthiness conditions

Before we formalise the relationship between a program's variables and the ALTG, it is worth introducing a healthiness condition to ensure that both the *altg* and *altg′* graphs are healthy (as defined in Section 3.4).

$$\begin{array}{lcl} \mathrm{HC1}_{\mathrm{U}}(Q) & \widehat{=} & Q = (\ Q \wedge \mathrm{HC}_{\mathrm{U}}(altg) \wedge \mathrm{HC}_{\mathrm{U}}(altg')\ ) \\ \mathrm{HC}_{\mathrm{U}}(G) & \widehat{=} & \mathrm{HC1}_{\mathrm{G}}(G) \wedge \mathrm{HC2}_{\mathrm{G}}(G) \wedge \mathrm{HC3}_{\mathrm{G}}(G) \wedge \mathrm{HC4}_{\mathrm{G}}(G) \end{array}$$

The first step in formalising the link between the graph and program variables is by insisting that the first-level nodes within the graph correspond precisely to the UTP program variables other than the model variables (i.e. *altg* and *altg′*).

$$\begin{array}{lcl} \mathrm{HC2a}_{\mathrm{U}}(Q) & \widehat{=} & Q = (\ Q \wedge \{\text{'}x \mid \text{'}x \in inv\alpha Q\} = \mathsf{labels}_{\mathrm{G}}(altg, \varnothing)\ ) \\ \mathrm{HC2b}_{\mathrm{U}}(Q) & \widehat{=} & Q = (\ Q \wedge \{\text{'}x \mid \text{'}x' \in outv\alpha Q\} = \mathsf{labels}_{\mathrm{G}}(altg', \varnothing)\ ) \\ \mathrm{HC2}_{\mathrm{U}}(Q) & \widehat{=} & \mathrm{HC2a}_{\mathrm{U}}(Q) \wedge \mathrm{HC2b}_{\mathrm{U}}(Q) \end{array}$$

where: $inv\alpha Q$ and $outv\alpha Q$ represent the input and output alphabets of program Q except for the model variables *altg* and *altg′* respectively; and the child labels of graph path are defined by $\mathsf{labels}_{\mathrm{G}}(G, p) \mathrel{\widehat{=}} \{l \mid P.l \in \mathsf{children}(G, \mathsf{node}_{\mathrm{G}}(G, p))\}$.

The second, and last, step in formalising the link between the graph and program variables is to ensure that the value of a variable is the same as the value stored within the ALTG.

$$\begin{array}{lcl} \mathrm{HC3a}_{\mathrm{U}}(Q) & \widehat{=} & Q = (\ Q \wedge (\forall\, x \mid \text{'}x \in inv\alpha Q \bullet x = *_{\mathrm{G}}(altg, \text{'}x))\ ) \\ \mathrm{HC3b}_{\mathrm{U}}(Q) & \widehat{=} & Q = (\ Q \wedge (\forall\, x \mid \text{'}x' \in outv\alpha Q \bullet x = *_{\mathrm{G}}(altg', \text{'}x))\ ) \\ \mathrm{HC3}_{\mathrm{U}}(Q) & \widehat{=} & \mathrm{HC3a}_{\mathrm{U}}(Q) \wedge \mathrm{HC3b}_{\mathrm{U}}(Q) \end{array}$$

4.2 Operations

Due to space limitations, we only present those operations that significantly differ from those of the standard UTP relational model, as presented in Chapter 2 of [3]: specifically, the assignment and program variable management operations.

The assignment operation is broken down into three cases, depending on the type of the r-value (i.e. the value to be assigned). These mirror the three cases presented in the trace-based graph model, except that the location is always

defined in terms of a possibly empty path from a UTP program variable. It is defined as follows:

$$
\begin{array}{lcl}
x.p := nlv & \widehat{=} & \mathrm{HCs_U}(altg' = ((altg, \text{'}x.p) :=_{\mathrm{G}} nlv)) \\
x.p := ecv & \widehat{=} & \mathrm{HCs_U}(altg' = ((altg, \text{'}x.p) :=_{\mathrm{G}} ecv)) \\
x.p := y.p_1 & \widehat{=} & \mathrm{HCs_U}(altg' = ((altg, \text{'}x.p) :=_{\mathrm{G}} \text{'}y.p_1))
\end{array}
$$

where $x.\varnothing = x$ and $\mathrm{HCs_U}(Q) \mathrel{\widehat{=}} \mathrm{HC3_U}(\mathrm{HC2_U}(\mathrm{HC1_U}(Q)))$. Note that the combined healthiness condition ensures that the consequences of updating shared values can be seen by all participating UTP program variables.

The variable introduction and elimination operations are also defined in terms of their effects on the ALTG. Here the variable introduction operation provides a default unset value to the introduced variable; and the variable elimination operation removes all references to the value from the graph.

$$
\begin{array}{lcl}
\mathsf{var}\, x & \widehat{=} & \exists\, x \bullet \mathrm{HCs_U}(altg' = ((altg, \text{'}x) :=_{\mathrm{G}} \text{¿})) \\
\mathsf{end}\, x & \widehat{=} & \exists\, x' \bullet \mathrm{HCs_U}(altg' = \mathsf{unlink_G}(altg, \text{'}x))
\end{array}
$$

4.3 Full abstraction

The ALTG-based UTP model of locations, outlined here, is fully abstract in the sense described earlier: two concrete location graphs are equivalent, as defined in Section 2.3, iff their corresponding ALTGs are equal. This is essentially because the underpinning ALTG model is fully abstract by design; it removes the need for explicitly indexed shareable locations. Here, each location has precisely one path-set that represents it.

5 Related Work

Our model of locations was inspired by Hoare and He's trace-based model of pointers [4]. It introduces the notion of a containable location. This significantly complicates the — already non-trivial — notion of assignment, which in [4] is defined in terms of *swinging* the pointer of the assigned location to its new contents. In our model, several contained pointers can be swung at once by an assignment operation, as the contents of:

- a containable location are duplicated on assignment;
- a shareable location are referenced (shared) on assignment;
- a location can include many shareable and containable locations.

The benefit of this extra complexity is that our location model enables the atomicity of assignment to be directly specified (or supported). For example, the copying of a `struct` in C++ or a `record` in Pascal can be captured.

Schieder has also adapted Hoare and He's work on trace-based pointers to provide a weakest precondition semantics for pointers [8]. Here, the object maps have been totalised in order to avoid undefinedness; this leads to the null pointer

being modelled by a node that has outbound edges (all of which point to itself). However, like [4] it does not support the notion of a containable location.

Cavalcanti, Harwood, and Woodcock have an *entity group* [6] inspired model of pointers and records [1]. Here, an entity group contains the set of path names that can be used to access the same value, where a path name is either:

1. a *simple name* of a UTP user variable; or
2. a *rooted field name*, which is a simple name extended by a dot-separated sequence of record field labels.

This notion of a path is similar to the one we use, in the sense that both use dot-separated labels to define a route from a given starting point to a location. The main difference is that every location in [1] is potentially shareable, whereas only some of the locations within our model are shareable. Specifically, [1] does not support the notion of a containable location.

A further difference between our model and those of [4], [8] and [1] is that we have an explicit notion of a pointer value (i.e. sharable location), as represented by a path-set of the form '$P.\$$'. One consequence of this is that our model directly supports the notion of a *handle*, which is a pointer to a pointer. Here the second pointer value (path-set) includes a path of the form '$p.\$.\$$'.

6 Conclusions

6.1 Summary

This paper augments the general relational model of the UTP with an Abstract Location Trace Graph (ALTG), which enables complex relationships between locations and their data to be represented. Here, both shareable and containable locations are modelled by a path-set. They differ in that only shareable locations can be dereferenced by a pointer, whose value is the shareable location's path-set itself. The key point is that containable locations actually contain rather than reference their contents, thus when they are copied their contents are duplicated rather than referenced. This mirrors situations where the whole of a compound value, such as a Pascal `record` or a C^{++} `struct`, is duplicated on assignment. In general, being able to control the amount of data that gets duplicated on assignment provides a means for directly supporting different levels of containment within a data structure.

One consequence of modelling a pointer's value as the path-set that defines its location is that it is possible to directly represent the concept of a handle (i.e. a pointer to a pointer). The combination of having direct support for contained locations and pointer values mirrors the features of our UTP model of objects [9]; it is what led to the development of this model from [4] and [1].

Overall, we argue that a general UTP model of pointers ought to consider both shareable and containable locations. Such models will provide support for languages like C$^{\#}$ (and our UTP object model), which have language constructs for building containable locations and handles.

6.2 Future work

In this paper we have presented both concrete and abstract models of locations. What we have not done is prove that the two models are consistent. We have also not shown how this model of locations can be applied to either UTP designs or objects [3, 9]. Finally, we have not considered the issues of:

- location ownership and encapsulation (e.g. as presented in [2, 5, 7]);
- location typing (e.g. augment a location with the type of its contents);
- location visibility (e.g. augment a location with read-only or scope modifiers).

Augmenting the UTP model of locations to handle any of these issues is left as future work.

References

1. Ana Cavalcanti, Will Harwood, and Jim Woodcock. Pointers and records in the unifying theories of programming. In Steve Dunne and Bill Stoddart, editors, *First International Symposium on Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
2. Daev Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, October 2002.
3. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Computer Science. Prentice Hall, 1998.
4. C.A.R. Hoare and Jifeng He. A trace model for pointers and objects. In 13^{th} *European Conference on Object-Oriented Programming*, pages 1–17, 1999.
5. James Noble, Daev Clarke, and John Potter. Object ownership for dynamic alias protection. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, 1999.
6. R.F. Paige and J.S. Ostroff. Erc: an object-oriented refinement calculus for Eiffel. *Formal Aspects of Computing*, 16(1):51–79, April 2004.
7. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In 17^{th} *Annual IEEE Symposium on Logic in Computer Science*, 2002.
8. Birgit Schieder. Pointer theory and weakest preconditions without addresses and heap. In Dexter Kozen and Carron Shankland, editors, *MPC 2004,* 7^{th} *International Conference on the Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 357–380. Springer, 2004.
9. Michael Anthony Smith and Jeremy Gibbons. Unifying Theories of Objects. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods*, volume 4591 of *Lecture Notes in Computer Science*, pages 599–618. Springer-Verlag, July 2007.