Improving Performance of Heterogeneous Agents

Fatma Özcan, V.S. Subrahmanian
University of Maryland, Dept. of CS
College Park, MD 20752, USA
{fatma,vs}@cs.umd.edu
and
Jürgen Dix
The University of Manchester, Dept. of CS
Oxford Road, Manchester, M13 9PL, UK
dix@cs.man.ac.uk

With the increase in agent-based applications, there are now agent systems that support *concurrent* client accesses. The ability to process large volumes of simultaneous requests is critical in many such applications. In such a setting, the traditional approach of serving these requests one at a time via queues (e.g. FIFO queues, priority queues) is insufficient. Alternative models are essential to improve the performance of such *heavily loaded* agents. In this paper, we propose a set of *cost-based algorithms* to *optimize* and *merge* multiple requests submitted to an agent. In order to merge a set of requests, one first needs to identify commonalities among such requests. First, we provide an *application independent framework* within which an agent developer may specify relationships (called *invariants*) between requests. Second, we provide two algorithms (and various accompanying heuristics) which allow an agent to automatically rewrite requests so as to avoid redundant work—these algorithms take invariants associated with the agent into account. Our algorithms are independent of any specific agent framework. For an implementation, we implemented both these algorithms on top of the *IMPACT* agent development platform, and on top of a (non-*IMPACT*) geographic database agent. Based on these implementations, we conducted experiments and show that our algorithms are considerably more efficient than methods that use the A^* algorithm.

Categories and Subject Descriptors: I.2.12 [**Artificial Intelligence**]: Distributed AI—*Intelligent Agents*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving; D.2.12 [**Software Engineering**]: Interoperability; H.2.4 [**Database Management**]: Heterogenous Databases

General Terms: Multi-Agency, Logical Foundations, Programming

Additional Key Words and Phrases: Heterogenous Data Sources, Multi-Agent Reasoning

1. MOTIVATION AND INTRODUCTION

A *heavily loaded* agent is one that experiences a large volume of service requests and/or has a large number of conditions to track on behalf of various users. The traditional model for servicing requests is via one kind of queue or the other (e.g. FIFO, LIFO, priority queue, etc.). For instance, a company may deploy a *PowerPoint* agent `ppt` that automatically creates PowerPoint presentations for different users

---

The first and third authors gratefully acknowledge support from the Army Research Laboratory under contract number DAAL01-97-K0135, and by DARPA/AFRL under grant number F306029910552.

based on criteria they have registered earlier. The finance director may get the latest budget data presented to him, a shop worker may get information on the latest work schedules for him, and the CEO may get information on stock upheavals.

If the `ppt` agent has thousands of such presentations to create for different users, it may well choose to exploit "redundancies" among the various requests to enhance its own performance. Hence, rather than sequentially creating a presentation for the CEO, then one for the finance director, then one for the marketing manager, then one for the shop manager, etc., it may notice that the finance director and CEO both want some relevant financial data—this data can be accessed and a PowerPoint page created for it once, instead of twice. Likewise, a heterogeneous database agent `hdb` tracking inventory information for thousands of users may well wish to exploit the commonality between queries such as

> *Find all suppliers who can provide 1000 automobile engines by June 25, 2003* and *Find all suppliers who can provide 1500 VX2 automobile engines by June 21, 2003.*

In this case, the latter query can be executed by using the answer returned by the first query, rather than by executing the second query from scratch. This may be particularly valuable when the `hdb` agent has to access multiple remote supplier databases—by leveraging the *common aspects* of such requests, the `hdb` agent can greatly reduce load on the network and the time taken to jointly process these two requests.

The same problem occurs in yet another context. [Subrahmanian et al. 2000] have described a framework called *IMPACT* within which software agents may be built on top of arbitrary data structures and software packages. In their framework, an agent manipulates a set of data structures (including a message box) via a set of well defined functions. The state of the agent at a given point in time consists of a set of objects in the agent's data structures. The agent also has a set of integrity constraints. When the agent state changes (this may happen if a message is received from another agent, a shared workspace is written by another agent or entity, a clock tick occurs, etc.), the agent must take some actions that cause the state to again be consistent with the integrity constraints. Hence, each agent has an associated set of actions (with the usual preconditions and effects), and an *agent program* which specifies under what conditions an agent is permitted to take an action, under what conditions it is obliged to take an action, under what conditions it is forbidden from taking an action, and under what conditions an action is in fact taken. [Eiter et al. 2000] have shown how (under some restrictions) such an agent program may be *compiled* into a set of conditions to be evaluated at run-time over the agent's state. When the agent state changes, then for each action α, one such condition needs to be evaluated over the state in order to determine which instances of that action (if any) need to be performed. Hence, numerous such conditions need to be simultaneously evaluated so that the agent can decide what actions to take so as to restore consistency of the state with the integrity constraints.

Therefore, in this paper, we consider the following technical problem. Suppose an agent is built on top of heterogeneous data structures (e.g. using methods such as those described in various agent frameworks such as [Eiter et al. 1999; Subrahmanian et al. 2000; Dix et al. 2000; Dix et al. 2001; Dix et al. 2000]).

> *Suppose the agent is confronted with a set S of requests. How should the agent process these requests so as to reduce the overall load on itself?*

In the case of the `ppt` agent for example, this capability will allow the agent to recognize the fact that many presentations requested by different clients require common financial data to be computed and/or analyzed, and hence, performing this *once* instead of *many times* will most certainly enhance performance. Likewise, in the case of the `hdb` agent, merging the two queries about automobile engines presented earlier allows the agent to reduce load on itself, thus allowing it to respond to other queries faster than by queuing.

The paper is organized as follows: First, we provide the basic definitions and some preliminary results that will be employed throughout the paper in Section 2. Then, we present our architecture in Section 3. In Sections 4 and 5, we discuss the development phase and the deployment phase components, respectively. The experiments are discussed in Section 6. Finally, Section 7 presents related work and Section 8 concludes the paper.

2. PRELIMINARIES

All agents manipulate some set $\mathcal{T}$ of data types and manipulate these types via some set of functions (application program interface functions). The input/output types of functions are known. If `d` is the name of a data structure (or even a software package), and f is an n-ary function defined in that package, then

$$\mathtt{d}:f(a_1,\ldots,a_n)$$

is a *code call.* This code call says

> *Execute function f as defined in data structure/package* `d` *on the stated list of arguments.*

We assume this code call returns as output, a *set* of objects—if an atomic object is returned, it can be coerced into a set. For instance, if we consider a commonly used data structure called a *quad-tree* [Samet 1989] for geographic reasoning, $\mathtt{quadtree}:range(\langle 20,30\rangle,\mathtt{T},40))$ may be a code call that says *find all objects in the quadtree the root of which is pointed to by* `T` *which are within 40 units of location* $\langle 20,30\rangle$—this query returns a set of points.

An *atomic code call condition* is an expression of the form

$$\mathbf{in}(t,\mathtt{d}:f(a_1,\ldots,a_{\mathtt{n}}))$$

which succeeds if t is in the set of answers returned by the code call in question. For example, $\mathbf{in}(t,\mathtt{excel}:chart(excelFile,rec,date))$ is an atomic code call condition that succeeds if t is a chart plotting *rec* with respect to *date* in the *excelFile.*

We assume that for each type τ manipulated by the agent, there is a set $root(\tau)$ of "root" variable symbols ranging over τ. In addition, suppose τ is a complex record type having fields $\mathtt{f_1},\ldots,\mathtt{f_n}$. Then, for every variable `X` of type τ, we require that $\mathtt{X.f_i}$ be a variable of type τ_i where τ_i is the type of field $\mathtt{f_i}$. In the same vein, if $\mathtt{f_i}$ itself has a sub-field `g` of type γ, then $\mathtt{X.f_i.g}$ is a variable of type γ, and so on. The variables, $\mathtt{X.f_i}$, $\mathtt{X.f_i.g}$, etc. are called *path variables.* For any path variable `Y` of the form `X.path`, where `X` is a root variable, we refer to `X` as the root of `Y`, denoted by $\mathit{root}(\mathtt{Y})$; for technical convenience, $\mathit{root}(\mathtt{X})$, where $\mathtt{X}$ is a root variable, refers to itself. If S is a set of variables, then $\mathit{root}(\mathtt{S}) = \{\mathit{root}(\mathtt{X}) \mid \mathtt{X} \in S\}$.

Convention 2.1. From now on, we use lower case letters $(a, b, c, c_1, \ldots)$ to denote constants and upper case letters $(\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \mathtt{X}_1, \ldots)$ to denote variables. When it is clear from context, we will also use lower case letters like s, t as metavariables ranging over constants, variables or terms.

A *code call condition* (ccc) may now be defined as follows:

(1) Every atomic code call condition is a code call condition.
(2) If s and t are either variables or objects, then $s = t$ is an (equality) code call condition.
(3) If s and t are either integers/real valued objects, or are variables over the integers/reals, then $s < t$, $s > t$, $s \leq t$, and $s \geq t$ are (inequality) code call conditions.
(4) If χ_1 and χ_2 are code call conditions, then $\chi_1 \,\&\, \chi_2$ is a code call condition.

Code call conditions provide a simple, but powerful language syntax to access heterogeneous data structures and legacy software code.

Example 2.1. [Sample ccc] The code call condition

$$\begin{array}{l}\mathbf{in}(\mathtt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{date}, "=", "11/15/99"))\ \& \\ \mathtt{FinanceRec.sales} \geq 10K\ \& \\ \mathbf{in}(\mathtt{C}, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \mathtt{FinanceRec}, \mathit{day}))\ \& \\ \mathbf{in}(\mathtt{Slide}, \mathtt{ppt}: \mathit{include}(\mathtt{C}, "\mathrm{presentation.ppt}"))\end{array}$$

is a complex condition that accesses and merges data across a relational database, an Excel file, and a PowerPoint file. It first selects all financial records associated with "11/15/99": this is done with the variable `FinanceRec` in the first line. It then filters out those records having sales more than $10K$ (second line). Using the remaining records, an Excel chart is created with day of sale on the x-axis and the resulting chart is included in the PowerPoint file "presentation.ppt" (fourth line).

In the above example, it is very important that the first code call be evaluable. If, for example the constant $\mathit{financeRel}$ were a variable, then

$$\mathtt{rel}: \mathit{select}(\mathtt{FinanceRel}, \mathit{date}, "=", "11/15/99")$$

would not be evaluable, unless there were another condition instantiating this variable. In order to come up with a notion of *evaluability*, we need the following notion.

Definition 2.2 (Dependent ccc's). For an atomic code call condition of the form $\mathbf{in}(\mathtt{X_i}, \mathtt{cc_i})$ *we define* $\mathit{root}(\mathtt{cc_i}) = \{\mathit{root}(\mathtt{Y}) \mid \mathtt{Y}$ *occurs in* $cc_i\}$ *and* $\mathit{root}(\mathtt{X_i}) = \{\mathit{root}(\mathtt{Y})) \mid \mathtt{Y}$ *occurs in* $\mathtt{X_i}\}$. *For an (in-)equality code call condition* $ccc_{in/eq}$ *we define* $\mathit{var}(ccc_{in/eq}) = \{\mathit{root}(\mathtt{Y})) \mid \mathtt{Y}$ *occurs in* $ccc_{in/eq}\}$.

A code call condition χ_j *is said to be* dependent on χ_i *iff the following holds:*

(1) ***Case 1:*** χ_i *is of the form* $\mathbf{in}(\mathtt{X_i}, \mathtt{cc_i})$.
  (a) If χ_j *is an atomic code call condition of the form* $\mathbf{in}(\mathtt{X_j}, \mathtt{cc_j})$ *then* $\mathit{root}(\mathtt{X_i}) \subseteq \mathit{root}(\mathtt{cc_j})$.

  (b) *If* χ_j *is an equality or inequality code call condition of the form* $\mathtt{s_1}$ *op* $\mathtt{s_2}$*, then either* $\mathtt{s_1}$ *is a variable and* $root(\mathtt{s_1}) \in root(\mathtt{X_i})$ *or* $\mathtt{s_2}$ *is a variable and* $root(\mathtt{s_2}) \in root(\mathtt{X_i})$ *or both.*

(2) ***Case 2:*** χ_i *is an (in-)equality code call condition.*

  (a) *If* χ_j *is an atomic code call condition of the form* $\mathbf{in}(\mathtt{X_j}, \mathtt{cc_j})$ *then* $var(\chi_i) \subseteq root(\mathtt{cc_j})$*.*

  (b) *If* χ_j *is an equality or inequality code call condition of the form* $\mathtt{s_1}$ *op* $\mathtt{s_2}$*, then either* $\mathtt{s_1}$ *is a variable and* $root(\mathtt{s_1}) \in var(\chi_i)$ *or* $\mathtt{s_2}$ *is a variable and* $root(\mathtt{s_2}) \in var(\chi_i)$ *or both.*

Example 2.3. [Dependency among ccc's] The ccc $\chi_1 : \mathtt{FinanceRec.sales} \geq 10K$ is dependent on the atomic code call condition

$$\chi_2 : \mathbf{in}(\mathtt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{date}, \mathtt{"="}, \mathtt{"11/15/99"})),$$

because $root(\mathtt{FinanceRec.sales}) \in root(\mathtt{FinanceRec})$. Similarly, the atomic code call condition $\chi_3 : \mathbf{in}(\mathtt{C}, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \mathtt{FinanceRec}, \mathit{day}))$ is dependent on the atomic code call condition χ_2, as the root variable `FinanceRec` which appears as an argument in the code call of χ_3 is instantiated in χ_2.

Definition 2.4 (Code Call Evaluation Graph (cceg) of a ccc). A code call evaluation graph *for a code call condition* $\chi = \chi_1 \& ... \& \chi_n$*,* $n \geq 1$ *where each* χ_i *is either an atomic, equality or inequality code call condition, is a directed graph* $cceg(\chi) = (V, E)$ *where:*

(1) $V =_{def} \{\chi_i \mid 1 \leq i \leq n\}$,

(2) $E =_{def} \{\langle \chi_i, \chi_j \rangle \mid \chi_j \textit{ is dependent on } \chi_i \textit{ and } 1 \leq i \neq j \leq n\}$.

Example 2.5. Figure 1 shows an example code call evaluation graph for the code call condition of Example 2.1.

If *finRel* were a variable `FinRel`, then the ccc would depend on the equality ccc $\mathtt{FinRel} = \mathit{finRel}$.

Using the dependency relation on the constituents of a code call condition, we are now able to give a precise description of an *evaluable* ccc.

Definition 2.6 (Evaluability of a ccc, $var_{base}(ccc)$*). A code call evaluation graph is* evaluable *iff*

(1) *It is acyclic.*

(2) *For all nodes* χ_i*, with in-degree* 0 *the following holds:*

  (a) *If* χ_i *is an atomic code call condition of the form* $\mathbf{in}(\mathtt{X_i}, \mathtt{d}: f(\mathtt{d_1}, \ldots, \mathtt{d_n}))$*, then each* $d_i, 1 \leq i \leq n$*, is ground.*

  (b) *If* χ_i *is an equality or inequality code call condition of the form* $\mathtt{s_1}$ *op* $\mathtt{s_2}$*, then either* $\mathtt{s_1}$ *or* $\mathtt{s_2}$ *or both are constants.*

A code call condition ccc is evaluable *iff it has an evaluable code call evaluation graph.*

For an evaluable ccc, we denote by $var_{base}(ccc)$ *the set of all variables ocurring in nodes having in-degree* 0*. The set* $var(ccc)$ *of all variables ocurring in ccc may be a superset of* $var_{base}(ccc)$*.*

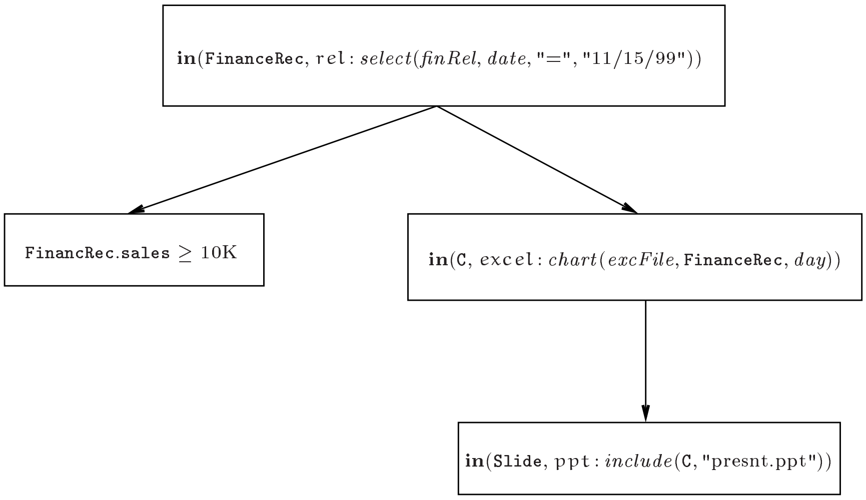

Fig. 1. The code call evaluation graph of Example 2.1

Example 2.7. The code call evaluation graph in Figure 1 is evaluable because the atomic code call condition of the only node with in-degree 0 has ground arguments in its code call and it contains no cycles.

In [Eiter et al. 2000] the notion of a *safe* code call was defined to provide the necessary means to check if a given code call is evaluable. It defines a linear ordering of atomic, equality and inequality code calls within a given code call condition in such a way that when executed from left to right the code call condition is executable. Before tying our new notion of *graph-evaluability* to the notion of *safety*, we recapitulate the definition of safety from [Eiter et al. 2000].

Definition 2.8 (Safe Code Call (Condition)).
A code call $\mathrm{d} : f(arg_1, \ldots, arg_n)$ *is* safe *iff each* arg_i *is ground. A code call condition* $\chi_1 \,\&\, \ldots \,\&\, \chi_n$, $n \geq 1$, *is* safe *iff there exists a permutation* π *of* $\chi_1, \ldots, \chi_n$ *such that for every* $i = 1, \ldots, n$ *the following holds:*

(1) If $\chi_{\pi(i)}$ *is an equality/inequality* $\mathtt{s}_1$ *op* $\mathtt{s}_2$*, then*
- *at least one of* $\mathtt{s}_1$*,*$\mathtt{s}_2$ *is a constant or a variable* $\mathtt{X}$ *such that root*$(\mathtt{X})$ *belongs to* $RV_{\pi(i)} = \{root(\mathtt{Y}) \mid \exists j < i \text{ s.t. } Y \text{ occurs in } \chi_{\pi(j)}\}$*;*
- *if* $\mathtt{s_i}$ *is neither a constant nor a variable* $\mathtt{X}$ *such that root*$(\mathtt{X}) \in RV_{\pi(i)}$*, then* $\mathtt{s_i}$ *is a root variable.*

(2) If $\chi_{\pi(i)}$ *is an atomic code call condition of the form* $\mathbf{in}(\mathtt{X}_{\pi(\mathtt{i})}, \mathtt{cc}_{\pi(\mathtt{i})})$*, then the root of each variable* $\mathtt{Y}$ *occurring in* $\mathtt{cc}_{\pi(i)}$ *belongs to* $RV_{\pi(i)}$*, and either* $\mathtt{X}_{\pi(\mathtt{i})}$ *is a root variable, or root*$(\mathtt{X}_{\pi(\mathtt{i})})$ *is from* $RV_{\pi(i)}$*.*

We call the permutation π *with the above properties a* witness to the safety.

Intuitively, a code call is safe, if we can reorder the atomic code call conditions occurring in it in a way such that we can evaluate these atoms left to right, assuming that root variables are incrementally bound to objects.

Example 2.9. Consider the code call condition

$$\mathbf{in}(\texttt{FinanceRec}, \texttt{rel}: \mathit{select}(\mathit{financeRel}, \texttt{data}, \texttt{"="}, \texttt{"11/15/99"}))\ \&$$
$$\mathbf{in}(\texttt{C}, \texttt{excel}: \mathit{chart}(\mathit{excelFile}, \texttt{FinanceRec}, \mathit{day})).$$

This code call condition is safe as it meets both of the safety requirements. However, the following code call condition is not safe:

$$\mathbf{in}(\texttt{FinanceRec}, \texttt{rel}: \mathit{select}(\mathit{financeRel}, \texttt{data}, \texttt{"="}, \texttt{"11/15/99"}))\ \&$$
$$\mathbf{in}(\texttt{C}, \texttt{excel}: \mathit{chart}(\texttt{ExcelFile}, \texttt{FinanceRec}, \mathit{day})).$$

This is because, there is no permutation of these two atomic code call conditions which allows safety requirement 1 to be met for the variable `ExcelFile`.

As a cceg is acyclic for any evaluable graph, ccegs determine a partial ordering $\preceq$ on the χ_i's:

$$\chi_i \preceq \chi_j \textit{ if and only if } \langle \chi_i, \chi_j \rangle \in E.$$

Hence, we may abuse notation and talk about *topological sorts* [Knuth 1997] of a graph to mean the topological sort of the associated partial ordering. Recall that given a partially ordered set $(S, \leq)$, a topological sorting of that set yields a linearly ordered set $(S, \preceq)$ such that $(\forall x, y \in S) x \leq y \rightarrow x \preceq y$. In the same vein, a topological sort of a directed acyclic graph (dag) is a linear ordering of nodes in the graph, such that if there exists an edge $\langle v_1, v_2 \rangle$ in the graph, then v_1 precedes v_2 in the topological sort.

Theorem 2.10. π is a witness to the safety of χ if and only if π is a valid topological sort of the cceg of χ.

The algorithm **Create-cceg** (Figure 2) takes a code call condition χ and creates an evaluable code call evaluation graph if χ is evaluable—otherwise it returns **NIL**.

The following example demonstrates the working of this algorithm for the code call condition of Example 2.1.

Example 2.11. Let

χ_1 : $\mathbf{in}(\texttt{FinanceRec}, \texttt{rel}: \mathit{select}(\texttt{financialRel}, \mathit{date}, \texttt{"="}, \texttt{"11/15/99"}))$,
χ_2 : $\texttt{FinanceRec.sales} \geq 10K$,
χ_3 : $\mathbf{in}(\texttt{C}, \texttt{excel}: \mathit{chart}(\mathit{excelFile}, \texttt{FinanceRec}, \mathit{day}))$, and
χ_4 : $\mathbf{in}(\texttt{Slide}, \texttt{ppt}: \mathit{include}(\texttt{C}, \texttt{"presentation.ppt"}))$.

First, $L = \{\chi_1, \chi_2, \chi_3, \chi_4\}$, $L' = Var = E = \emptyset$. We first create a node for each of the four code call conditions. $Ok = \{\chi_1, \chi_2\}$, as all arguments in the code call of χ_1 are ground, and 10K is a constant in χ_2. Next, we create the edge (χ_1, χ_2). Because χ_2 depends on χ_1. Then, $L = \{\chi_3, \chi_4\}$, $L' = \{\chi_1, \chi_2\}$ and $Var = \{\texttt{FinanceRec}\}$. In the first iteration of the while loop $\Psi = \{\chi_3\}$ as χ_3 depends on χ_1, and all variables in χ_3 (`FinanceRec`) are in Var. Var becomes $\{\texttt{FinanceRec}, \texttt{C}\}$ and we create the edge (χ_1, χ_3). Now, $L = \{\chi_4\}$, $L' = \{\chi_1, \chi_2, \chi_3\}$. In the second iteration of the while loop $\Psi = \{\chi_4\}$, since χ_4 depends on χ_3 and all variables in χ_4 (namely $\{\texttt{C}\}$) are in Var. This time, Var becomes $\{\texttt{FinanceRec}, \texttt{C}, \texttt{Slide}\}$, and we add the edge $\langle \chi_3, \chi_4 \rangle$ to the graph. Now L becomes the empty set and the algorithm returns the code call evaluation graph given in Figure 1.

```
Create-cceg($\chi$)

/* Input:  $\chi$ : $\chi_1 \& \chi_2 \& ... \& \chi_n$                    */
/* Output:NIL, if $\chi$ is not evaluable                     */
/*          a cceg $CCEG = (V, E)$, if $\chi$ is evaluable*/

$L$    $:= \{\chi_1, \chi_2, .., \chi_n\}$;
$L'$   $:= \emptyset$;
$Var$  $:= \emptyset$;
$E$    $:= \emptyset$;
$V$    $:= \{\chi_i \mid\ 1 \leq i \leq n\}$;
$Ok$   $:= \{\chi_i \mid$  $\chi_i$ is either of the form in(X, d:$f$(args)) where args is ground or
                of the form s$_1$ $op$ s$_2$, where either s$_1$ or s$_2$ or both are constants };
for all pairs $\langle \chi_i, \chi_j \rangle$, $\chi_i, \chi_j \in Ok$ such that $\chi_j$ is dependent on $\chi_i$
create an edge $\langle \chi_i, \chi_j \rangle$ and add it to $E$;
$Var$ $:= Var \cup \{root(\mathtt{X_i}) \mid \mathbf{in}(\mathtt{X_i}, \mathrm{d}:f(\mathtt{args})) \in Ok\}$;
$L$    $:= L - Ok$;
$L'$   $:= L' \cup Ok$;
while ($L$ is not empty) do
    $\Psi := \{\chi_i \mid\ \chi_i \in L$ and all variables in $\chi_i$ are in $Var$ and
              $\exists \chi_j \in L‘$ such that $\chi_i$ depends on $\chi_j\}$;
    if card $(\Psi) = 0$ then Return NIL;
    else
    $Var := Var \cup \{root(\mathtt{X_i}) \mid \mathbf{in}(\mathtt{X_i}, \mathrm{d}:f(\mathtt{args})) \in \Psi\}$;
    for all pairs $\langle \chi_i, \chi_j \rangle$, $\chi_j \in \Psi$, such that $\chi_j$ is dependent on $\chi_i \in L'$
    create an edge $\langle \chi_i, \chi_j \rangle$ and add it to $E$;
    $L := L - \Psi$;
    $L' := L' \cup \Psi$;

Return $(V, E)$;
End-Algorithm
```

Fig. 2. **Create-cceg** Algorithm

Convention 2.2. Throughout the rest of this paper, we assume that all code call conditions considered are evaluable and that the graph associated with each code call condition has been generated.

The **Create-cceg** algorithm runs in $O(n^3)$ time, where n is the number of constituents χ_i of χ. The number of iterations of the while loop is bounded by n, and the body of the while loop can be executed in quadratic time.

We have conducted experiments to evaluate the execution time of the **Create-cceg** algorithm. Those experiments are described in detail in Section 6.1.

Definition 2.12 (State of an agent). The state of an agent is a set of ground code call conditions.

When an agent developer builds an agent, she specifies several parameters. One of these parameters must include some *domain-specific* information, explicitly laying out what inclusion and equality relations are known to hold of code calls. Such information is specified via *invariants.*

Definition 2.13 (Invariant Expression).

- *Every evaluable code call condition is an invariant expression. We call such expressions* atomic*.*
- *If* ie_1 *and* ie_2 *are invariant expressions, then* $(\mathsf{ie}_1 \cup \mathsf{ie}_2)$ *and* $(\mathsf{ie}_1 \cap \mathsf{ie}_2)$ *are invariant expressions. (We will often omit the parentheses.)*

Example 2.14. Two examples of invariant expressions are:

$$\begin{array}{l}\mathbf{in}(\mathtt{StudentRec}, \mathtt{rel}: \mathit{select}(\mathit{courseRel}, \mathit{exam}, \texttt{"="}, \mathit{midterm1})) \,\&\\ \mathbf{in}(\mathtt{C}, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \mathtt{StudentRec}, \mathit{grade}))\end{array}$$

$$\begin{array}{l}\mathbf{in}(\mathtt{X}, \mathtt{spatial}: \mathit{horizontal}(\mathtt{T}, \mathtt{B}, \mathtt{U})) \cup (\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{horizontal}(\mathtt{T}', \mathtt{B}', \mathtt{U}')) \cup\\ \mathbf{in}(\mathtt{Z}, \mathtt{spatial}: \mathit{horizontal}(\mathtt{T}', \mathtt{B}', \mathtt{U}))).\end{array}$$

What is the meaning, i.e. the *denotation* of such expressions? The first invariant represents the set of all objects c such that

$$\begin{array}{l}\mathbf{in}(\mathtt{StudentRec}, \mathtt{rel}: \mathit{select}(\mathit{courseRel}, \mathit{exam}, \texttt{"="}, \mathit{midterm1})) \,\&\\ \mathbf{in}(c, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \mathtt{StudentRec}, \mathit{grade}))\end{array}$$

holds: we are looking for instantiations of `C`. Note that under this viewpoint, the intermediate variable `StudentRec` which is needed in order to instantiate `C` to an object c does not matter. There might just as well be situations where we are interested in pairs $\langle c, \mathit{studentrec}\rangle$ instead of just c. Therefore a notion of denotation must be flexible enough to allow this.

Let us now consider the invariant

$$\begin{array}{l}\mathbf{in}(\mathtt{StudentRec}, \mathtt{rel}: \mathit{select}(\mathit{courseRel}, \mathit{exam}, \texttt{"="}, \mathtt{TypeofExam})) \,\&\\ \mathbf{in}(\mathtt{C}, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \mathtt{StudentRec}, \mathit{grade}))\end{array}$$

where the object *midterm1* has been replaced by the variable `TypeofExam` which is now a base variable. Then we might be interested in all c's that result if an instantiation of `TypeofExam` is given, i.e. for different instantiations of `TypeofExam` we get different c's. Thus we have to distinguish carefully between various sorts of variables: *base* variables (defined in Definition 2.17), *auxiliary* variables and the *main* variables defining the set of objects of interest.

Definition 2.15 (Denotation of an Invariant Expression). Let ie *be an invariant expression with* $\mathit{var}(\mathsf{ie}) = \mathit{var}_{base}(\mathsf{ie}) \cup \{\mathtt{V}_1, \ldots, \mathtt{V}_n\}$*. The denotation of* ie *with respect to a state* S*, an assignment* θ *of the variables in* $\mathit{var}_{base}(\mathsf{ie})$ *and a sequence* $\langle \mathtt{V}_{i_1}, \ldots, \mathtt{V}_{i_k}\rangle$ *(where* $\mathtt{V}_{i_1}, \ldots, \mathtt{V}_{i_k}\} \subseteq \{\mathtt{V}_1, \ldots, \mathtt{V}_n\}$*) is defined as follows:*

- *Let*

$$[\mathsf{ie}]_{S,\theta} := \{\, \langle o_{\pi(1)}, \ldots, o_{\pi(n_k)}\rangle \mid \begin{array}{l}(\mathsf{ie}\theta)\tau \textit{ is ground and is true in state } S,\\ \pi \textit{ is a permutation on } \{1, \ldots, n\},\ n_k \leq n,\\ \tau \textit{ is a grounding substitution,}\\ \tau \textit{ is of the form } [\mathtt{V}_1/o_1, \ldots, \mathtt{V}_n/o_n]\ \}\end{array}$$

- $[\mathsf{ie}_1 \cap \mathsf{ie}_2]_{S,\theta} := [\mathsf{ie}_1]_{S,\theta} \cap [\mathsf{ie}_2]_{S,\theta}$*,*
- $[\mathsf{ie}_1 \cup \mathsf{ie}_2]_{S,\theta} := [\mathsf{ie}_1]_{S,\theta} \cup [\mathsf{ie}_2]_{S,\theta}$*.*

The variables in $\{\mathtt{V}_{\pi(1)}, \ldots, \mathtt{V}_{\pi(n_k)}\}$ are called *main variables* while all remaining variables $\{\mathtt{V}_{\pi(n_k+1)}, \ldots, \mathtt{V}_{\pi(n)}\}$ are called *auxiliary.* The substitution τ is defined on the set of *main variables* (in our example above it is the set $\{\mathtt{C}\}$). The set of auxiliary variables consists of $\{\texttt{StudentRec}\}$ and the only base variable is `TypeofExam`. Taking the first viewpoint in our example above, τ would be defined on $\{\mathsf{C}, \texttt{StudentRec}\}$.

As usual, we abuse notation and say that $\mathsf{ie}_1 \subseteq \mathsf{ie}_2$ if $[\mathsf{ie}_1]_{S,\theta} \subseteq [\mathsf{ie}_2]_{S,\theta}$ for all S and all assignments θ. Similarly, we say that $\mathsf{ie}_1 = \mathsf{ie}_2$ if $[\mathsf{ie}_1]_{S,\theta} = [\mathsf{ie}_2]_{S,\theta}$ for all S and all assignments θ. Now we are ready to define an invariant.

Definition 2.16 (Invariant Condition (ic)). An invariant condition atom *is a statement of the form t_1 Op t_2 where Op $\in \{\leq, \geq, <, >, =\}$ and each of t_1, t_2 is either a variable or a constant. An invariant condition (IC) is defined inductively as follows:*

(1) Every invariant condition atom is an ic.

(2) If C_1 and C_2 are ic's, then $C_1 \wedge C_2$ and $C_1 \vee C_2$ are ic's.

Definition 2.17 (Invariant inv, var_{base}(inv), INV_{simple}, $INV_{ordinary}$, INV). An invariant*, denoted by inv, is a statement of the form*

$$ic \Longrightarrow \quad ie_1 \;\; \Re \;\; ie_2 \tag{1}$$

where

(1) ic is an invariant condition, all variables occuring in ic are among $var_{base}(ie_1) \cup var_{base}(ie_2)$.

(2) $\Re \in \{=, \subseteq\}$, and

(3) ie_1, ie_2 are invariant expressions.

If ie_1 and ie_2 both contain solely atomic *code call conditions, then we say that inv is a simple invariant.*

If ic is a conjunction of invariant condition atoms, then we say that inv is an ordinary invariant.

We denote by var_{base}(inv) the set of all variables of inv that need to be instantiated in order to evaluate inv in the current state: $var_{base}(inv) := var_{base}(ie_1) \cup var_{base}(ie_2)$.

The set of all invariants is denoted by INV. The set of all simple invariants is denoted by INV_{simple} and the set of all ordinary invariants is denoted by $INV_{ordinary}$.

An invariant expresses *semantic knowledge* about a domain. Invariants used by each of our two example agents—ppt and hdb are given below.

Example 2.18. The following are valid invariant conditions: $val_1 \leq val_2$, $\texttt{Rel}_1 = \texttt{Rel}_2$. Note that such expressions can be evaluated over a given state S. Only the two relations $\leq$ and $\geq$ require that the constants occurring on the right or left hand sides must be of the appropriate type: these relations must be defined over each state S.

The invariant

$$\begin{gathered}\texttt{File} = \texttt{File}' \wedge \texttt{Rec} = \texttt{Rec}' \wedge \texttt{Col} = \texttt{Col}' \\ \Longrightarrow \\ \mathbf{in}(\mathsf{C}, \mathrm{excel}: \mathit{chart\_one}(\texttt{File}, \texttt{Rec}, \texttt{Col})) = \mathbf{in}(\mathsf{C}', \mathrm{excel}: \mathit{chart\_two}(\texttt{File}', \texttt{Rec}', \texttt{Col}'))\end{gathered}$$

says that these two code call conditions are equivalent if their arguments unify. Note that the code calls involved are different. The invariant,

$$\begin{array}{c}\texttt{Rel} = \texttt{Rel}' \wedge \texttt{Attr} = \texttt{Attr}' \wedge \texttt{Op} = \texttt{Op}' = \texttt{"}\leq\texttt{"} \wedge \texttt{Val} < \texttt{Val}' \\ \Longrightarrow \\ \mathbf{in}(\texttt{X}, \mathtt{rel}: \mathit{select}(\texttt{Rel}, \texttt{Attr}, \texttt{Op}, \texttt{Val})) \subseteq \mathbf{in}(\texttt{Y}, \mathtt{rel}: \mathit{select}(\texttt{Rel}', \texttt{Attr}', \texttt{Op}', \texttt{Val}'))\end{array}$$

says that the code call condition $\mathbf{in}(\texttt{X}, \mathtt{rel}: \mathit{select}(\texttt{Rel}, \texttt{Attr}, \texttt{Op}, \texttt{Val}))$ can be evaluated by using the results of the code call condition

$$\mathbf{in}(\texttt{Y}, \mathtt{rel}: \mathit{select}(\texttt{Rel}', \texttt{Attr}', \texttt{Op}', \texttt{Val}'))$$

if the above conditions are satisfied. Note that this expresses *semantic information* that is not available on the syntactic level: the operator "$\leq$" is related to the relation symbol "$<$".

Convention 2.3. Throughout the rest of this paper, we assume that we have the code calls $\mathtt{ag}: \mathit{addition}(\texttt{X}, \texttt{Y})$ and $\mathtt{ag}: \mathit{subtraction}(\texttt{X}, \texttt{Y})$ available for all agents $\mathtt{ag}$. These code calls return the sum, (resp. the difference) of $\texttt{X}$ and $\texttt{Y}$, where $\texttt{X}$ and $\texttt{Y}$ range over the reals or the integers. We also assume we have code calls $\mathtt{ag}: \mathit{ge}_0(X)$ (resp. $\mathtt{ag}: \mathit{geq\_0}(X)$) available which returns 1 if $\texttt{X}$ is strictly greater (resp. greater or equal) than 0 and 0 otherwise.

By stating invariants, we focus interest on states where the invariants hold. This is like in classical predicate logic, where we write down axioms and thereby constrain the set of models—we are only interested in the class of models satisfying the axioms. We therefore have to define formally what it means for a state S to satisfy an invariant inv.

Definition 2.19 (Satisfaction, $S \models \mathsf{inv}$, $\mathcal{I} \models \mathsf{inv}$, Taut).
A state S satisfies *the invariant inv having the form shown in Formula (1) above with respect to an assignment θ iff for every ground instance $(\mathsf{inv}\theta)\tau$ of $\mathsf{inv}\theta$, it is the case that either $(\mathsf{ic}\theta)\tau$ evaluates to false, or $(\mathsf{ie}_1\,\theta)\tau\,\Re\,(\mathsf{ie}_2\,\theta)\tau$ is true in S.*

We say that a set of invariants $\mathcal{I}$ entails *an invariant inv iff all states S and assignments θ satisfying $\mathcal{I}$ also satisfy inv. We write $\mathcal{I} \models \mathsf{inv}$. We call an invariant inv a tautology, if inv is true in all states S for all assignments θ:*

$$\mathit{Taut} =_{def} \{\mathsf{inv} \mid\ \models \mathsf{inv}\}.$$

From now on we do not mention explicitly the assignment θ and we write simply $S \models \mathsf{inv}$.

It is worth noting that there are indeed trivial invariants that are satisfied in all states: such invariants are like tautologies in classical logic (therefore their name in the last definition). For example the following invariant is true in all states whatsoever (note the difference from the similar invariant above):

$$\begin{array}{c}\texttt{File} = \texttt{File}' \wedge \texttt{Rec} = \texttt{Rec}' \wedge \texttt{Col} = \texttt{Col}' \Longrightarrow \\ \mathbf{in}(\texttt{C}, \mathtt{excel}: \mathit{chart}(\texttt{File}, \texttt{Rec}, \texttt{Col})) = \mathbf{in}(\texttt{C}', \mathtt{excel}: \mathit{chart}(\texttt{File}', \texttt{Rec}', \texttt{Col}'))\end{array}$$

The reason that this last invariant is a tautology, is that for the same set of instances of $\texttt{Y}$ for a code call $\mathtt{d}: f(\texttt{Y})$, we always get the same set representing the atomic code call condition $\mathbf{in}(\texttt{X}, \mathtt{d}: f(\texttt{Y}))$.

Theorem 2.20.
There is a translation $\mathfrak{Trans}$ which associates with each conjunction ic of invariant

condition atoms, and invariant expression ie *another invariant expression* $\mathfrak{Trans}(\mathsf{ic},\mathsf{ie}_1)$ *such that the following holds for all states* S*, assignments* θ *and invariants* $\mathsf{ic} \Longrightarrow \mathsf{ie}_1 \ \Re\ \mathsf{ie}_2$

$$(S,\theta) \models (\mathsf{ic} \Longrightarrow \mathsf{ie}_1 \ \Re\ \mathsf{ie}_2)$$
$$\textit{if and only if}$$
$$(S,\theta) \models \textit{true} \Longrightarrow \mathfrak{Trans}(\mathsf{ic},\mathsf{ie}_1) \ \Re\ \mathfrak{Trans}(\mathsf{ic},\mathsf{ie}_2).$$

Corollary 2.21 (Eliminating Invariant Conditions, $\mathfrak{Trans}$*). Let* $\mathsf{inv} : \mathsf{ic} \Longrightarrow \mathsf{ie}_1 \ \Re\ \mathsf{ie}_2$ *be an arbitrary invariant. Then, the following holds for all states* S *and assignments* θ

$$(S,\theta) \models (\mathsf{ic} \Longrightarrow \mathsf{ie}_1 \ \Re\ \mathsf{ie}_2)$$
$$\textit{if and only if}$$
$$(\forall C_i, 1 \leq i \leq m) \quad (S,\theta) \models \textit{true} \Longrightarrow \mathfrak{Trans}(C_i,\mathsf{ie}_1) \ \Re\ \mathfrak{Trans}(C_i,\mathsf{ie}_2).$$

where the C_i*,* $1 \leq i \leq m$*, are the disjuncts in the DNF of* ic*.*

3. ARCHITECTURE

Let us suppose now that we have a set $\mathcal{I}$ of invariants, and a set $\mathcal{S}$ of data structures that are manipulated by the agent. How exactly should a set $\mathcal{C}$ of code call conditions be merged together? And what needs to be done to support this? Our architecture contains two parts:

(i) a *development time* phase stating what the agent developer must specify when building her agent, and what algorithms are used to operate on that specification, and

(ii) a *deployment time* phase which specifies how the above development-time specifications are used when the agent is in fact running autonomously.

We describe each of these pieces below.

3.1 Development Time Phase

When the agent developer builds her agent, the following things need to be done.

(1) First, the agent developer specifies a set $\mathcal{I}$ of invariants.

(2) Suppose $\mathcal{C}$ is a set of CCCs to be evaluated by the agent. Each code call condition $\chi \in \mathcal{C}$ is represented via an evaluable cceg. Let $\mathit{INS}(\mathcal{C})$ represent the set of all nodes in ccegs of χs in $\mathcal{C}$. That is,

$$\mathit{INS}(\mathcal{C}) = \{v_i \mid \exists \chi \in \mathcal{C} \ s.t.\ v_i \ is\ in\ \chi' s\ cceg\}.$$

This can be done by a topological sort of the cceg for each $\chi \in \mathcal{C}$.

(3) Additional invariants can be derived from the initial set $\mathcal{I}$ of invariants. This requires the ability to check whether a set $\mathcal{I}$ of invariants implies an inclusion relationship between two invariant expressions. We will provide a *generic* test called **Chk_Imp** for implication checking between invariants. Although the set of invariants entailed by $\mathcal{I}$ is defined by Definition 2.19, the set of invariants actually derived by the **Chk_Imp** test will depend on the set of axioms used in the test. Hence, some **Chk_Imp** tests will be sound, but not complete. On the other hand, some tests will be "more complete" than others, because the

set of invariants derived by them will be a superset of the set of invariants derived by others. "More complete" tests may use a larger set of axioms, hence will be more expensive to compute. The agent developer can select a test that is appropriate for her agent. Given an arbitrary (but fixed) **Chk_Imp** test, we will provide an algorithm called **Compute-Derived-Invariants** that calculates the set of derivable invariants from the initial set $\mathcal{I}$ of invariants and needs to be executed just once.

3.2 Deployment Time Phase

Once the agent has been "developed" and deployed and is running, it will need to continuously determine how to merge a set $\mathcal{C}$ of code call conditions. This will be done as follows:

(1) The system must identify three types of relationships between nodes in $\mathit{INS}(\mathcal{C})$.

Identical ccc's: First, we'd like to identify nodes $\chi_1, \chi_2 \in \mathit{INS}(\mathcal{C})$ which are "equivalent" to one another, i.e. $\chi_1 = \chi_2$ is a logical consequence of the set of invariants $\mathcal{I}$. This requires a definition of *equivalence* of two code call conditions w.r.t. a set of invariants. This strategy is useful because we can replace the two nodes χ_1, χ_2 by a single node. This avoids redundant computation of both χ_1 and χ_2.

Implied ccc's: Second, we'd like to identify nodes $\chi_1, \chi_2 \in \mathit{INS}(\mathcal{C})$ which are not equivalent in the above sense, but such that either $\chi_1 \subseteq \chi_2$ or $\chi_2 \subseteq \chi_1$ hold, but not both. Suppose $\chi_1 \subseteq \chi_2$. Then we can compute χ_2 first, and then compute χ_1 from the answer returned by computing χ_2. This way of computing χ_1, χ_2 may be faster than computing them separately.

Overlapping ccc's: Third, we'd like to identify nodes $\chi_1, \chi_2 \in \mathit{INS}(\mathcal{C})$ for which the preceding two conditions do not hold, but $\chi_1 \,\&\, \chi_2$ is consistent with $\mathit{INS}(\mathcal{C})$. In this case, we might be able to compute the answer to $\chi_1 \vee \chi_2$. From the answer to this, we may compute the answer to χ_1 and the answer to χ_2. This way of computing χ_1, χ_2 may be faster than computing them separately.

We will provide an algorithm, namely **Improved-CSI**, which will use the set of derived invariants returned by the **Compute-Derived-Invariants** algorithm above, to detect commonalities (equivalent, implied and overlapping code call conditions) among members of $\mathcal{C}$.

Example 3.1. The two code call conditions $\mathbf{in}(\mathtt{X}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}, \mathtt{L}, \mathtt{R}))$ and $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{L}', \mathtt{R}'))$ are equivalent to one another if their arguments are unifiable. The results of evaluating the code call condition

$$\mathbf{in}(\mathtt{Z}, \mathtt{spatial}: \mathit{range}(\mathtt{T}, 40, 50, 25))$$

is a subset of the results of evaluating the code call condition

$$\mathbf{in}(\mathtt{W}, \mathtt{spatial}: \mathit{range}(\mathtt{T}', 40, 50, 50))$$

if $\mathtt{T} = \mathtt{T}'$. Note that $\mathtt{spatial}: \mathit{range}(\mathtt{T}, \mathtt{X}, \mathtt{Y}, \mathtt{Z})$ returns all points in $\mathtt{T}$ that are $\mathtt{Z}$ units away from the point $\langle \mathtt{X}, \mathtt{Y} \rangle$. In this case, we can compute the results of the former code call condition by executing a selection on the results of the latter rather than executing the former from scratch. Finally, consider the following two code call conditions:

$$\mathbf{in}(\mathtt{X}, \mathtt{spatial}: \mathit{horizontal}(\mathit{map}, 100, 200)),$$
$$\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{horizontal}(\mathit{map}, 150, 250)).$$

Here $\mathtt{spatial}: \mathit{horizontal}(\mathit{map}, a, b)$ returns all points ($\mathtt{X}$, $\mathtt{Y}$) in *map* such that $a \leq \mathtt{Y} \leq b$. Obviously, the results of neither of these two code call conditions are subset of the results of the other. However, the results of these two code call conditions *overlap* with one another. In this case, we can execute the code call condition $\mathbf{in}(\mathtt{Z}, \mathtt{spatial}: \mathit{horizontal}(\mathit{map}, 100, 250))$. Then, we can compute the results of the two code call conditions by executing selections on the results of this code call condition.

(2) We will then provide two procedures to merge sets of code call conditions, **BFMerge** and **DFMerge**, that take as input, *(i)* the set $\mathcal{C}$ and *(ii)* the output of the **Improved-CSI** algorithm above, and *(iii)* a cost model for agent code call condition evaluations. Both these algorithms are parameterized by heuristics and we propose three alternative heuristics. Then we evaluate our six implementations (3 heuristics times 2 algorithms) and also compare it with an A^* based approach.

4. DEVELOPMENT PHASE

Prior to deployment of the agent, once the agent developer has defined a set of invariants, we compute a set of derived invariants from it. These derived invariants are stored. Once deployed, when the agent is confronted with a set of requests from other agents, it can examine these stored derived invariants for a "pattern match" which then enables it to classify invariants into one of the three categories listed (equivalent, implied or overlapping invariants).

Consider the case when $\mathcal{I}$ contains the two invariants:

$$\mathtt{V}_1 \leq \mathtt{V}_2 \implies \mathbf{in}(\mathtt{X}, \mathrm{d}_1 : f_1(\mathtt{V}_1)) = \mathbf{in}(\mathtt{Y}, \mathrm{d}_2 : f_2(\mathtt{V}_2)). \quad (2)$$
$$\mathtt{V}_3 \leq \mathtt{V}_4 \implies \mathbf{in}(\mathtt{Z}, \mathrm{d}_2 : f_2(\mathtt{V}_3)) \subseteq \mathbf{in}(\mathtt{W}, \mathrm{d}_3 : f_3(\mathtt{V}_4)). \quad (3)$$

Clearly from these two invariants, we can infer the invariant

$$\mathtt{V}_1 \leq \mathtt{V}_2 \wedge \mathtt{V}_2 \leq \mathtt{V}_4 \implies \mathbf{in}(\mathtt{X}, \mathrm{d}_1 : f_1(\mathtt{V}_1)) \subseteq \mathbf{in}(\mathtt{W}, \mathrm{d}_3 : f_3(\mathtt{V}_4)) \quad (4)$$

Algorithm **Combine_1** (Figure 3) combines two invariants. The algorithm uses a **simplify** routine which simplifies a conjunction of invariant conditions and checks if the resulting invariant condition is inconsistent or not. If so, it returns **NIL**. The **Combine_1** algorithm makes use of two important algorithms: **Chk_Imp** and **Chk_Ent**, which we will discuss in detail later. The **Chk_Imp** algorithm checks if one invariant expression implies another, while the **Chk_Ent** algorithm checks if some member of a set of invariants entails an other invariant. Let us first define the table which is implemented by the **Combine_1** algorithm.

Definition 4.1 (Combine1). Let $\mathsf{inv}_1 : \mathsf{ic}_1 \implies \mathsf{ie}_1\ \Re_1\ \mathsf{ie}'_1$ *and* $\mathsf{inv}_2 : \mathsf{ic}_2 \implies \mathsf{ie}_2\ \Re_2\ \mathsf{ie}'_2$*. Then, the following table provides the resulting derived relation of the form*

$$\boldsymbol{simplify}(\mathsf{ic}_1 \wedge \mathsf{ic}_2) \implies \mathsf{ie}_1\ \Re_1\ \mathsf{ie}'_2$$

when inv_1 *and* inv_2 *are combined. The "*" denotes a "don't care" condition in this table. The* ***simplify*** *routine checks whether* $\mathsf{ic}_1 \wedge \mathsf{ic}_2$ *is inconsistent. If so, it returns*

$\Re_1$	$\Re_2$	**Chk_Imp**$(\mathsf{ie}_1', \mathsf{ie}_2)$	**Chk_Imp**$(\mathsf{ie}_2, \mathsf{ie}_1')$	derived_rel
*	*	False	*	**NIL**
$=$	$=$	True	True	$=$
$=$	$\subseteq$	True	*	$\subseteq$
$\subseteq$	$=$	True	*	$\subseteq$
$\subseteq$	$\subseteq$	True	*	$\subseteq$

Table 1. Summary of Combining Two Invariants

false*, if not it returns an equivalent (perhaps simplified) formula for* $\mathsf{ic}_1 \wedge \mathsf{ic}_2$ *(the precise realization of* ***simplify*** *is not important here and leaves enough freedom for the actual implementation):*

$$\textbf{\textit{simplify}}(\mathit{ic}) = \begin{cases} \text{true}, & \textit{if for all } (S,\theta) \models \mathit{ic}, \\ \text{false}, & \textit{if for all } (S,\theta) \models \neg \mathit{ic}, \\ \phi^1, & \textit{otherwise.} \end{cases}$$

Figure 3 implements a slightly generalized version of the last definition. Namely, we assume that there is given a set $\mathcal{I}$ of invariants and we are considering states satisfying these invariants. This is an additional parameter. For simplicity, assume that $\Re_1 = \Re_2 =$ "$\subseteq$". The idea is that although the subset relation $\mathsf{ie}_1' \subseteq \mathsf{ie}_2$ might not hold in general (i.e. in all states) it could be implied by the invariants in $\mathcal{I}$ (i.e. holds in all states satisfying $\mathcal{I}$). That is, if there is $\mathsf{ic}^* \implies \mathsf{ie}_1^* \subseteq \mathsf{ie}_2^* \in \mathcal{I}$ s.t. $(\mathsf{ic}^{**} \implies \mathsf{ie}_1' \subseteq \mathsf{ie}_1^* \in \mathcal{I}$, $\mathsf{ic}^{***} \implies \mathsf{ie}_2^* \subseteq \mathsf{ie}_2 \in \mathcal{I}$, and $(\mathsf{ic}_1 \wedge \mathsf{ic}_2) \rightarrow (\mathsf{ic}^* \wedge \mathsf{ic}^{**} \wedge \mathsf{ic}^{***}))$. Under these conditions, we can derive the invariant **simplify**$(\mathsf{ic}_1 \wedge \mathsf{ic}_2) \implies \mathsf{ie}_1 \subseteq \mathsf{ie}_2'$.

We introduce three notions, **Chk_Imp**, **Chk_Taut** and **Chk_Ent** of increasing complexity. The first notion, **Chk_Imp**, is a relation between invariant expressions.

Definition 4.2 (Implication: ***Chk_Imp****,* $\mathit{ie}_1 \rightarrow \mathit{ie}_2$*).*
An invariant expression ie_1 *is said to imply another invariant expression* ie_2*, denoted by* $\mathit{ie}_1 \rightarrow \mathit{ie}_2$*, iff it is the case that* $[\mathit{ie}_1]_{S,\theta} \subseteq [\mathit{ie}_2]_{S,\theta}$ *for all* S *and all assignments* θ*.*

Chk_Imp *is said to be an* implication check *algorithm if it takes two invariant expressions* $\mathit{ie}_1, \mathit{ie}_2$ *and returns a boolean output. We say that* ***Chk_Imp*** *is* sound *iff whenever* ***Chk_Imp****(*$\mathit{ie}_1, \mathit{ie}_2$*)=true, then* ie_1 *implies* ie_2*. We say* ***Chk_Imp*** *is* complete *iff* ***Chk_Imp***$(\mathit{ie}_1, \mathit{ie}_2) =$ *true if and only if* ie_1 *implies* ie_2*.*

If ***Chk_Imp***$_1$*,* ***Chk_Imp***$_2$ *are both sound, and for all* $\mathit{ie}_1, \mathit{ie}_2$*,* ***Chk_Imp***$_1(\mathit{ie}_1, \mathit{ie}_2) =$ *true implies that* ***Chk_Imp***$_2(\mathit{ie}_1, \mathit{ie}_2) =$ *true, then we say that* ***Chk_Imp***$_2$ *is* more complete *than* ***Chk_Imp***$_1$*.*

---

[1] where ϕ is any formula equivalent to ic, i.e. for all states (S,θ): $(S,\theta) \models \phi \leftrightarrow \mathsf{ic}$.

```
Combine_1 ($\mathsf{inv}_1, \mathsf{inv}_2, \mathcal{I}$)
/* $\mathsf{inv}_1 : \mathsf{ic}_1 \implies \mathsf{ie}_1 \; \Re_1 \; \mathsf{ie}'_1$ */
/* $\mathsf{inv}_2 : \mathsf{ic}_2 \implies \mathsf{ie}_2 \; \Re_2 \; \mathsf{ie}'_2$ */

if (Chk_Imp $(\mathsf{ie}'_1, \mathsf{ie}_2) = \mathit{false}$) and
    (there is no $\mathsf{ic}^* \implies \mathsf{ie}^*_1 \subseteq \mathsf{ie}^*_2 \in \mathcal{I}$ s.t.
    ($\mathsf{ic}^{**} \implies \mathsf{ie}'_1 \subseteq \mathsf{ie}^*_1 \in \mathcal{I}$,   $\mathsf{ic}^{***} \implies \mathsf{ie}^*_2 \subseteq \mathsf{ie}_2 \in \mathcal{I}$,
     $(\mathsf{ic}_1 \wedge \mathsf{ic}_2) \rightarrow (\mathsf{ic}^* \wedge \mathsf{ic}^{**} \wedge \mathsf{ic}^{***})$)), then
     Return NIL;
if ($\Re_1 = \Re_2 =$ "=") then
   if (Chk_Imp $(\mathsf{ie}_2, \mathsf{ie}'_1) = \mathit{true}$) or
    (there is $\mathsf{ic}^* \implies \mathsf{ie}^*_1 \subseteq \mathsf{ie}^*_2 \in \mathcal{I}$ s.t.
    ($\mathsf{ic}^{**} \implies \mathsf{ie}_2 \subseteq \mathsf{ie}^*_1 \in \mathcal{I}$,   $\mathsf{ic}^{***} \implies \mathsf{ie}^*_2 \subseteq \mathsf{ie}'_1 \in \mathcal{I}$,
     $(\mathsf{ic}_1 \wedge \mathsf{ic}_2) \rightarrow (\mathsf{ic}^* \wedge \mathsf{ic}^{**} \wedge \mathsf{ic}^{***})$))), then
     $\mathit{relation} := (\mathsf{ie}_1 = \mathsf{ie}'_2)$;
   else $\mathit{relation} := (\mathsf{ie}_1 \subseteq \mathsf{ie}'_2)$;
else $\mathit{relation} := (\mathsf{ie}_1 \subseteq \mathsf{ie}'_2)$;
$\mathit{derived}_i c :=$ simplify$(\mathsf{ic}_1 \wedge \mathsf{ic}_2)$;
if ($\mathit{derived}_i c = \mathit{false}$) then Return NIL;
$\mathit{derived\_inv} := (\mathit{derived}_i c \implies \mathit{relation})$;
if (there is $\mathsf{inv} \in \mathcal{I}$ with (Chk_Ent ($\mathsf{inv}$, $\mathit{derived\_inv}$) $= \mathit{true}$) then
     Return NIL;
else Return $\mathit{derived\_inv}$;
End-Algorithm
```

Fig. 3. **Combine_1** Algorithm

*Definition 4.3 (**Chk_Taut**, **Chk_Ent** as Relations between Invariants).* ***Chk_Taut*** *is said to be a* tautology check *algorithm if it takes a single invariant* inv *and returns a boolean output.* ***Chk_Taut*** *is* sound *iff whenever* ***Chk_Taut***(inv)*=true, then* $\mathsf{inv} \in$ *Taut (see Definition 2.19).* ***Chk_Taut*** *is* complete *iff* ***Chk_Taut***(inv) *= true if and only if* $\mathsf{inv} \in$ *Taut.*

Chk_Ent *is said to be an* entailment check *algorithm if it takes two invariants* $\mathsf{inv}_1, \mathsf{inv}_2$ *and returns a boolean output. We say that* ***Chk_Ent*** *is* sound *iff whenever* ***Chk_Ent***$(\mathsf{inv}_1, \mathsf{inv}_2)$*=true, then* inv_1 *entails* inv_2 $(\mathsf{inv}_1 \models \mathsf{inv}_2)$*. We say* ***Chk_Ent*** *is* complete *iff* ***Chk_Ent***$(\mathsf{inv}_1, \mathsf{inv}_2) = $ *true if and only if* inv_1 *entails* inv_2*.*

Similarly to Definition 4.2, we use the notion of being more complete *for tautology as well as for entailment check algorithms.*

*Lemma 4.4 (Relation between **Chk_Imp**, **Chk_Taut** and **Chk_Ent**).*

(1) ***Chk_Imp*** *can be reduced to* ***Chk_Taut****:*

$$\textbf{\textit{Chk\_Imp}}(\mathit{ie}_1, \mathit{ie}_2) \text{ if and only if } \textbf{\textit{Chk\_Taut}}(\mathit{true} \implies \mathit{ie}_1 \subseteq \mathit{ie}_2).$$

(2) ***Chk_Taut*** *can be reduced to* ***Chk_Imp****:*

$$\begin{gathered} \textbf{\textit{Chk\_Taut}}((C_1 \vee C_2 \vee \ldots \vee C_m) \implies \mathit{ie}_1 \subseteq \mathit{ie}_2) \\ \text{if and only if} \\ \forall C_i, 1 \leq i \leq m, \textbf{\textit{Chk\_Imp}}(\mathfrak{Trans}(C_i, \mathit{ie}_1), \mathfrak{Trans}(C_i, \mathit{ie}_2)). \end{gathered}$$

(3) ***Chk_Taut*** *is an instance of* ***Chk_Ent****.*

Thus in general, implication checking between invariant expressions is a special case of tautology checking of invariants. Conversely, checking tautologies is an instance of implication checking. Note that checking simple invariants is reduced to checking implications of non-simple invariant expressions.

It is also obvious that checking for tautologies is a special case of the entailment problem.

The following results tell us that the implementation of the **Chk_Imp** routine used in the **Combine_1** algorithm is undecidable in general. Even if we restrict to finite domains, it is still intractable.

*Proposition 4.5 (Undecidability of **Chk_Imp**, **Chk_Taut**, **Chk_Ent**).*
Suppose we consider arbitrary datatypes. Then the problem of checking whether an arbitrary invariant expression ie_1 *implies another invariant expression* ie_2 *is undecidable. The same holds for checking tautologies of invariants or entailment between invariants.*

*Proposition 4.6 (*co-NP *Completeness of Checking Implication).*
Suppose all datatypes have a finite domain (i.e. each datatype has only finitely many values of that datatype). Then the problem of checking whether an arbitrary invariant expression ie_1 *implies another invariant expression* ie_2 *is* co-NP *complete. The same holds for the problem of checking whether an invariant is a tautology.*

As the problem of checking implication (and hence equivalence) between invariant expressions is co-NP complete, in this paper, we decided to study the tradeoffs involved in using sound, but perhaps incomplete implementations of implication checking.

There are clearly many ways of implementing the algorithm **Chk_Imp** that are sound, but not complete. In this paper, we propose a generic algorithm to implement **Chk_Imp**, where the complexity can be controlled by two input parameters—an *axiomatic inference system* and a *threshold*.

—The *axiomatic inference system* used by **Chk_Imp** includes some axioms and inference rules. By selecting the axioms and inference rules, the agent developer is controlling the branching factor of the search space.

—The second parameter called the *threshold* is either an integer or ∞, and determines the maximum depth of the search tree. If it is ∞, then the generic algorithm does not have an upper bound on the number of rule applications, and terminates either when it proves the implication or there is no further rule that is applicable(i.e. *failure*). When it is an integer value, the algorithm reports *failure* if it cannot prove the implication by using the threshold number of rule applications.

We have conducted experiments with different instances of these two parameters. Those experiments are discussed in detail in Section 6.2.

It is important to note that the set of all derived invariants obtained from $\mathcal{I}$ may be very large because they contain "redundant" constraints. For instance, using our example $\mathcal{I}$ above, every invariant of the form

$$\begin{array}{c}\mathtt{V}_1 \leq \mathtt{V}_2 \ \wedge \mathtt{V}_2 \leq \mathtt{V}_4 \\ \Longrightarrow \\ \mathbf{in}(\mathtt{X}, \mathrm{d}_1 : f_1(\mathtt{V}_1)) \subseteq \mathbf{in}(\mathtt{W}, \mathrm{d}_3 : f_3(\mathtt{V}_4)) \cup \mathbf{in}(\mathtt{T}, \mathrm{d}_4 : f_4(\mathtt{V}_5)) \ldots\end{array}$$

would be entailed from $\mathcal{I}$—however, these invariants are *redundant* as they are entailed by the single invariant (4).

As we have seen above (Propositions 4.5, 4.6), such an entailment test between invariants is either undecidable or intractable. It would be much better if we had a purely syntactical test (which must be necessarily incomplete) of checking such implications.

The following lemma shows that entailment between two invariants can, under certain assumptions, be reduced to a syntactical test.

Lemma 4.7. Let $\mathsf{inv}_1 : \mathsf{ic}_1 \Longrightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}'_1$ *and* $\mathsf{inv}_2 : \mathsf{ic}_2 \Longrightarrow \mathsf{ie}_2 \subseteq \mathsf{ie}'_2$ *be two simple invariants, i.e.* ie_1 *has the form* $\mathbf{in}(\mathtt{X}, \mathtt{d}_1 : f_1(\ldots))$, ie'_1 *has the form* $\mathbf{in}(\mathtt{X}, \mathtt{d}'_1 : f'_1(\ldots))$, ie_2 *has the form* $\mathbf{in}(\mathtt{Y}, \mathtt{d}_2 : f_2(\ldots))$ *and* ie'_2 *has the form* $\mathbf{in}(\mathtt{Y}, \mathtt{d}'_2 : f'_2(\ldots))$.

If $\mathsf{inv}_1 \models \mathsf{inv}_2$ *and* inv_2 *is not a tautology* ($\not\models \mathsf{inv}_2$)*, then the following holds:*

(1) $\mathtt{d}_1 = \mathtt{d}_2$ *and* $f_1 = f_2$,

(2) $\mathtt{d}'_1 = \mathtt{d}'_2$ *and* $f'_1 = f'_2$,

(3) In all states that do not satisfy inv_2*, it holds "*$\mathsf{ic}_2 \to \mathsf{ic}_1$*". I.e. each counterexample for* inv_2 *is also a counterexample for* inv_1*.*

Corollary 4.8 (Sufficient Condition for ***Chk_Ent****).*
There is a sufficient condition for ***Chk_Ent***$(\mathsf{inv}_1, \mathsf{inv}_2)$ *based on* ***Chk_Imp*** *and* ***Chk_Taut****: First check whether* $\mathsf{inv}_2 \in$ *Taut. If yes,* ***Chk_Ent***$(\mathsf{inv}_1, \mathsf{inv}_2)$ *holds. If not, check whether* $\mathsf{ic}_2 \to \mathsf{ic}_1$ *holds in all states (i.e.* ***Chk_Imp***$(\mathsf{ic}_2, \mathsf{ic}_1)$*). If yes,* ***Chk_Ent***$(\mathsf{inv}_1, \mathsf{inv}_2)$ *holds.*

In this paper, we use the following sound but incomplete **Chk_Ent** algorithm. Let $\mathsf{inv}_1 : \mathsf{ic}_1 \Longrightarrow \mathsf{ie}_1 \ \Re_1 \ \mathsf{ie}'_1$ and $\mathsf{inv}_2 : \mathsf{ic}_2 \Longrightarrow \mathsf{ie}_2 \ \Re_2 \ \mathsf{ie}'_2$. Then, **Chk_Ent**$(\mathsf{inv}_1, \mathsf{inv}_2) =$ *true iff*

(1) For all states S: $S \models \mathsf{ic}_2 \to \mathsf{ic}_1$,
(2) ($\Re_1 =$ "$\subseteq$" and $\Re_2 =$ "$\subseteq$") or ($\Re_1 =$ "$=$" and $\Re_2 =$ "$\subseteq$"),
(3) $\mathsf{ie}_2 \to \mathsf{ie}_1$,
(4) $\mathsf{ie}'_1 \to \mathsf{ie}'_2$.

4.1 Computing All Derived Invariants

In this section, we define how given a set $\mathcal{I}$, the set of all invariants that are entailed by $\mathcal{I}$ may be computed using the selected **Chk_Imp** and **Chk_Ent** algorithms.

The **Compute-Derived-Invariants** algorithm presented in Figure 4 takes as input a set of invariants $\mathcal{I}$, and returns a set of invariants $\mathcal{I}^*$, such that every invariant in $\mathcal{I}^*$ is entailed by $\mathcal{I}$. Although the **Compute-Derived-Invariants** algorithm has exponential running time, it is executed only once at registration-time, and hence the worst case complexity of the algorithm is acceptable.

Lemma 4.9. For all $\mathcal{I}$: $\{\mathsf{inv}_1, \mathsf{inv}_2\} \cup \mathcal{I} \models$ ***Combine_1***$(\mathsf{inv}_1, \mathsf{inv}_2, \mathcal{I})$*.*

Combine_1 does not derive all invariants that are logically entailed by $\mathcal{I}$. For example from "true $\Rightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}_2$" and "true $\Rightarrow \mathsf{ie}_2 \subseteq \mathsf{ie}_1$" we can infer "true $\Rightarrow \mathsf{ie}_1 = \mathsf{ie}_2$". We call this procedure, slightly generalized, **Combine_2**. It is illustrated in Figure 5. The **unify** routine takes two invariant expressions and returns the most general unifier if the two are unifiable, and returns **NIL** if they are not unifiable.

```
Compute-Derived-Invariants($\mathcal{I}$)
$X := \mathcal{I}$;
$change := true$;
$Done := \emptyset$;
while $change$ do
  $change := false$
  forall $\mathsf{inv}_i \in X$ do
    forall $\mathsf{inv}_j \in X - \{\mathsf{inv}_i\}$ s.t. $(\mathsf{inv}_i, \mathsf{inv}_j) \notin Done$ do
      $derived\_inv_1 :=$ combine_1($\mathsf{inv}_i, \mathsf{inv}_j, X$);
      if $derived\_\mathsf{inv}_1$ != NIL then
        $X := X \cup \{derived\_inv_1\}$; $change := true$;
      $derived\_inv_2 :=$ combine_1($\mathsf{inv}_j, \mathsf{inv}_i, X$);
      if $derived\_inv_2$ != NIL then
        $X := X \cup \{derived\_inv_2\}$; $change := true$;
      $derived\_inv_3 :=$ combine_2($\mathsf{inv}_j, \mathsf{inv}_i$);
      if $derived\_inv_3$ != NIL then
        $X := X \cup \{derived\_inv_3\}$; $change := true$;
      $derived\_inv_4 :=$ combine_3($\mathsf{inv}_j, \mathsf{inv}_i$);
      if $derived\_inv_4$ != NIL then
        $X := X \cup \{derived\_inv_4\}$; $change := true$;
      $Done := Done \cup \{(\mathsf{inv}_i, \mathsf{inv}_j), (\mathsf{inv}_j, \mathsf{inv}_i)\}$;
Return $X$.
End-Algorithm
```

Fig. 4. **Compute-Derived-Invariants** Algorithm

```
Combine_2($\mathsf{inv}_1, \mathsf{inv}_2$)
/* $\mathsf{inv}_1 : \mathsf{ic}_1 \implies \mathsf{ie}_1 \ \Re_1 \ \mathsf{ie}'_1$ */
/* $\mathsf{inv}_2 : \mathsf{ic}_2 \implies \mathsf{ie}_2 \ \Re_2 \ \mathsf{ie}'_2$ */

if ($\Re_1 = \Re_2 =$ "$\subseteq$") then
  $\theta := unify(\mathsf{ie}_2, \mathsf{ie}'_1)$;
  $\gamma := unify(\mathsf{ie}'_2, \mathsf{ie}_1)$;
  if ($\theta$ != NIL) and ($\gamma$ != NIL) then
    $derived_i c :=$ simplify($(\mathsf{ic}_1 \wedge \mathsf{ic}_2)\theta\gamma$);
    if ($derived_i c = false$) then Return NIL;
    $derived\_inv := (derived_i c \Longrightarrow (\mathsf{ie}_1)\theta\gamma = (\mathsf{ie}'_1)\theta\gamma)$;
    Return $derived\_inv$;
  else Return NIL.
else Return NIL.
End-Algorithm
```

Fig. 5. **Combine_2** Algorithm

Another set we need is the set of all invariant tautologies

$$\text{Taut} =_{def} \{\text{true} \Rightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}_2 : \textbf{Chk\_Imp}(\mathsf{ie}_1, \mathsf{ie}_2)\}.$$

Obviously, all tautologies are satisfied in all states and the invariant computed in the **Combine_2** Algorithm (if it exists) is entailed by the invariants it is computed from.

Lemma 4.10. $\{\mathsf{inv}_1, \mathsf{inv}_2\} \models$ ***Combine_2***$(\mathsf{inv}_1, \mathsf{inv}_2)$.

However, the above sets are still not sufficient. Consider the situation $\mathsf{inv}_1 : x < 0 \implies \mathsf{ie}_1 \ \Re_1 \ \mathsf{ie}'_1$, and $\mathsf{inv}_2 : x \geq 0 \implies \mathsf{ie}_1 \ \Re_1 \ \mathsf{ie}'_1$. Then, we can conclude **true** $\implies \mathsf{ie}_1 \ \Re_1 \ \mathsf{ie}'_1$. However, neither **Combine_1** nor **Combine_2** is able to compute this invariant. As a result, we define the final routine, **Combine_3**, given in Figure 6, to capture these cases.

```
Combine_3($\mathsf{inv}_1, \mathsf{inv}_2$)
/* $\mathsf{inv}_1 : \mathsf{ic}_1 \implies \mathsf{ie}_1 \ \Re_1 \ \mathsf{ie}'_1$ */
/* $\mathsf{inv}_2 : \mathsf{ic}_2 \implies \mathsf{ie}_2 \ \Re_2 \ \mathsf{ie}'_2$ */

if ($\Re_1 = \Re_2$) then
  $\theta :=$ unify($\mathsf{ie}_1, \mathsf{ie}_2$);
  $\gamma :=$ unify($\mathsf{ie}'_1, \mathsf{ie}'_2$);
  if ($\theta$ != NIL) and ($\gamma$ != NIL) then
    $derived_ic :=$ simplify($(\mathsf{ic}_1 \vee \mathsf{ic}_2)\theta\gamma$);
    $derived\_inv := derived_ic \implies ((\mathsf{ie}_1)\theta\gamma \ \Re_1 \ (\mathsf{ie}'_1)\theta\gamma)$;
    Return $derived\_inv$;
Return NIL
End-Algorithm
```

Fig. 6. **Combine_3** Algorithm

We emphasize in the **Combine_3** Algorithm our use of the **simplify** routine introduced just after Definition 4.1. Our example is captured because $x < 0 \vee x \geq 0$ is simplified to true. By recursively applying **Combine_3**, one can also handle more complicated intervals like $x < 0 \ \vee \ (x \geq 0 \wedge x < 1) \ \vee \ x \geq 1$.

Lemma 4.11. $\{\mathit{inv}_1, \mathit{inv}_2\} \models$ ***Combine_3***$(\mathit{inv}_1, \mathit{inv}_2)$.

Definition 4.12 (Operator $C_{\mathcal{I}}$). We associate with any set $\mathcal{I}$ of invariants, a mapping $C_{\mathcal{I}} : \mathit{INV} \to \mathit{INV}$ which maps sets of invariants to sets of invariants, as follows:

$$\begin{array}{lll} C_{\mathcal{I}}(X) =_{def} & \{\textbf{\textit{Combine\_1}}(\mathit{inv}_1, \mathit{inv}_2, X \cup \mathcal{I}) & \mid \mathit{inv}_1, \mathit{inv}_2 \in X \cup \mathcal{I}\} \ \cup \\ & \{\textbf{\textit{Combine\_2}}(\mathit{inv}_1, \mathit{inv}_2) & \mid \mathit{inv}_1, \mathit{inv}_2 \in X \cup \mathcal{I}\} \ \cup \\ & \{\textbf{\textit{Combine\_3}}(\mathit{inv}_1, \mathit{inv}_2) & \mid \mathit{inv}_1, \mathit{inv}_2 \in X \cup \mathcal{I}\} \ \cup \\ & \mathcal{I} \cup X & . \end{array}$$

Definition 4.13 (Powers of $C_{\mathcal{I}}$). The powers of $C_{\mathcal{I}}(X)$ are defined as follows:

$$\begin{array}{ll} C_{\mathcal{I}} \uparrow^{0} & := \mathit{Taut} \\ C_{\mathcal{I}} \uparrow^{(i+1)} & := C_{\mathcal{I}}(C_{\mathcal{I}} \uparrow^{i}) \\ C_{\mathcal{I}} \uparrow^{\omega} & := \bigcup_{i \geq 0}(C_{\mathcal{I}} \uparrow^{i}) \end{array}$$

Proposition 4.14 (Monotonicity of $C_{\mathcal{I}}$). If $X_1 \subseteq X_2$, then $C_{\mathcal{I}}(X_1) \subseteq C_{\mathcal{I}}(X_2)$.

Lemma 4.15. $C_{\mathcal{I}}(C_{\mathcal{I}} \uparrow^{\omega}) \subseteq C_{\mathcal{I}} \uparrow^{\omega}$.

Lemma 4.16. $C_{\mathcal{I}} \uparrow^{\omega} \subseteq \{\mathit{inv} \mid \mathcal{I} \models \mathit{inv}\}$.

What we are really interested in is a *converse* of the last lemma, namely that all invariants that follow from $\mathcal{I}$ can be derived. Strictly speaking, this is not the case: we already noticed that there are many redundant invariants that follow from $\mathcal{I}$ but are subsumed by others. Such "redundant" invariants contribute little. We show below that whenever an invariant is entailed from $\mathcal{I}$ as a whole, it is already entailed by another variant in $C_{\mathcal{I}} \uparrow^{\omega}$. This is the statement of our main Corollary 4.18.

Theorem 4.17 (All Entailed "$\subseteq$"-Invariants are Subsumed in $C_{\mathcal{I}} \uparrow^{\omega}$). Suppose $\mathcal{I} \models \mathsf{inv}$. We assume further that all the invariants are simple and that $\Re =$ "$\subseteq$" in all invariants. Then, there is $\mathsf{inv}' \in C_{\mathcal{I}} \uparrow^{\omega}$ such that inv' entails inv.

Corollary 4.18 (All Entailed Invariants are Subsumed in $C_{\mathcal{I}} \uparrow^{\omega}$). We are now considering arbitrary simple invariants, i.e. $\Re = \{\subseteq, =\}$. If $\mathcal{I} \models \mathsf{inv}$, then there exists $\mathsf{inv}' \in C_{\mathcal{I}} \uparrow^{\omega}$ such that inv' entails inv.

The following corollary tells us that if the implementation of **Chk_Imp** and **Chk_Ent** algorithms used are complete, then the **Compute-Derived-Invariants** algorithm correctly computes all derived invariants.

*Corollary 4.19 (Development-Time Check). Suppose **Chk_Imp** is a complete implication check, and **Chk_Ent** is a complete subsumption check algorithm. Then, the set of invariants returned by the **Compute-Derived-Invariants** has the following properties:*

(1) Every invariant returned by it is implied by $\mathcal{I}$ and

*(2) If an invariant is implied by $\mathcal{I}$, then there is an invariant inv' returned by the **Compute-Derived-Invariants** algorithm that entails inv.*

Our results above apply to simple invariants only. The reason is that in Table 1 only a subset of all possible derivable invariants are listed. For example even if the **Chk_Imp** tests do not hold, then there are still the following nontrivial invariants entailed:

(1) derived-$\mathsf{inv}_1 : \mathsf{ic}_1 \wedge \mathsf{ic}_2 \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2)\ \Re\ (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$

(2) derived-$\mathsf{inv}_2 : \mathsf{ic}_1 \wedge \mathsf{ic}_2 \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2)\ \Re\ (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$

In fact, our framework can be easily extended as follows. Let $iv_1 : \mathsf{ic}_1 \Longrightarrow \mathsf{ie}_1\ \Re_1\ \mathsf{ie}'_1$ and $iv_2 : \mathsf{ic}_2 \Longrightarrow \mathsf{ie}_2\ \Re_2\ \mathsf{ie}'_2$. In addition to the derived invariant returned by the **Combine_1** algorithm, the new extended **XCombine_1** also returns the derived invariants determined by Tables 2 and 4.1.

5. DEPLOYMENT PHASE

Once an agent is *up and running*, it is continuously confronted with requests for its services. One crucial observation is that there might be enormous overlap among these requests. These overlaps can be exploited if a given set $\mathcal{C}$ of code call conditions are merged in a way that executes common portions only once. However, in order to exploit commonalities, we must first determine the type of those commonalities, that is we must first identify code call conditions (1) that are *equivalent* to other code call conditions, (2) that are *implied* by other code call conditions and (3) that *overlap* with other code call conditions. Moreover, given two code call conditions ccc_1 and ccc_2, it might be the case that they are neither equivalent, nor implied, nor overlapped. On the other hand, parts of ccc_1 and $ccc2$ maybe equivalent, implied or overlapped. We also want to exploit such cases. This gives rise to the following definition:

$\Re_1$	$\Re_2$	**simplify**$(\mathsf{ic}_1, \mathsf{ic}_2)$	derived_inv
$*$	$*$	**NIL**	**NIL**
$\subseteq$	$\subseteq$	ic'	$\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) \subseteq (\mathsf{ie}_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}'_1 \cap \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cap \mathsf{ie}_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}'_2) \subseteq (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$
$\subseteq$	$=$	ic'	$\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) = (\mathsf{ie}_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}'_1 \cap \mathsf{ie}_2) = (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cap \mathsf{ie}_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}'_2) \subseteq (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$
$=$	$\subseteq$	ic'	$\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) \subseteq (\mathsf{ie}_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}'_1 \cap \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) = (\mathsf{ie}'_1 \cap \mathsf{ie}_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}'_2) = (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$
$=$	$=$	ic'	$\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) = (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) = (\mathsf{ie}_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}'_1 \cap \mathsf{ie}_2) = (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}_2) = (\mathsf{ie}'_1 \cap \mathsf{ie}_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cap \mathsf{ie}'_2) = (\mathsf{ie}'_1 \cap \mathsf{ie}'_2)$

Table 2. **XCombine** for Arbitrary Invariants

Definition 5.1 (Sub-Code Call Condition). Let $ccc = \chi_1 \& \chi_2 \& \ldots \& \chi_n$ *be a code call condition.* $ccc^j := \chi_{i_1} \& \chi_{i_2} \& \ldots \& \chi_{i_j}$*, for* $1 \leq i_1, \ldots i_j \leq n$ *and* $i_l \neq i_k \ \forall 1 \leq l, k \leq j$ *is called a* sub-code call condition *of ccc.*

Example 5.2. Let $ccc_1 = \chi_1 \& \chi_2 \& \chi_3 \& \chi_4 \& \chi_5$. Then, $\chi_1 \& \chi_2 \& \chi_3$, $\chi_1 \& \chi_3 \& \chi_5$ and $\chi_2 \& \chi_5$ are some sub-code call conditions of ccc_1.

Note that a code call condition with k atomic/(in)equality code call conditions has 2^k different sub-code call conditions. We are now ready to define equivalent, implied and overlapping sub-code call conditions χ, χ'. To do so, we need to fix the variable(s) in each *ccc* to which we want to project (see Definition 2.15: the sequence $\langle v_{i_1}, \ldots, v_{i_k} \rangle$ of variables occurring in χ_1, and the sequence $\langle v'_{i_1}, \ldots, v'_{i_k} \rangle$ of variables occurring in χ_2 are important). Often the sequences consist just of one single variable: this is the case when there is only one non-base variable occurring in the ccc's. In that case we do not explicitly mention the sequences.

Definition 5.3 (Equivalent (Sub-) CCC). Two (sub-) code call conditions χ_1 *and* χ_2 *are said to be* equivalent *w.r.t. the sequences* $\langle v_{i_1}, \ldots, v_{i_k} \rangle$ *and* $\langle v'_{i_1}, \ldots, v'_{i_k} \rangle$*, denoted by* $\chi_1 \equiv \chi_2$*, if and only if for all states* S *of the agent and all assignments* θ*, it is the case that*

$$[\chi_1]_{S,\theta,\langle v_{i_1}, \ldots, v_{i_k} \rangle} = [\chi_2]_{S,\theta,\langle v'_{i_1}, \ldots, v'_{i_k} \rangle}.$$

In the case of equivalent ccc's, we only need to execute one of the sub-code call conditions. We can use the cached solutions for the other sub-code call condition.

Example 5.4. The ccc $\mathbf{in}(\mathtt{C}, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \mathtt{FinanceRec}, \mathit{day}))$ is equivalent wrt. the sequences $\langle C \rangle$, $\langle C' \rangle$ to the ccc $\mathbf{in}(\mathtt{C}', \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \mathtt{Rec}, \mathit{day}))$,

$\Re_1$	$\Re_2$	$\mathbf{simplify}(\mathsf{ic}_1, \mathsf{ic}_2)$	derived_inv
$*$	$*$	**NIL**	**NIL**
$\subseteq$	$\subseteq$	ic'	$\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) \subseteq (\mathsf{ie}_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}'_1 \cup \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cup \mathsf{ie}_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}'_2) \subseteq (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$
$\subseteq$	$=$	ic'	$\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) = (\mathsf{ie}_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}'_1 \cup \mathsf{ie}_2) = (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cup \mathsf{ie}_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}'_2) \subseteq (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$
$=$	$\subseteq$	ic'	$\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) \subseteq (\mathsf{ie}_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}'_1 \cup \mathsf{ie}_2) \subseteq (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) = (\mathsf{ie}'_1 \cup \mathsf{ie}_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}'_2) = (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$
$=$	$=$	ic'	$\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) = (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) = (\mathsf{ie}_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}'_1 \cup \mathsf{ie}_2) = (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}_2) = (\mathsf{ie}'_1 \cup \mathsf{ie}_2)$ $\mathsf{ic}' \Longrightarrow (\mathsf{ie}_1 \cup \mathsf{ie}'_2) = (\mathsf{ie}'_1 \cup \mathsf{ie}'_2)$

Table 3. **XCombine** for Arbitrary Invariants

since the two code call conditions unify with the mgU $\gamma = [\mathtt{FinanceRec}/\mathtt{Rec}]$.

Definition 5.5 (Implied (Sub-) CCC). A (sub-) code call condition χ_1 is said to imply *another (sub-) code call condition χ_2 wrt. the sequences $\langle v_{i_1}, \dots, v_{i_k} \rangle$ and $\langle v'_{i_1}, \dots, v'_{i_k} \rangle$, denoted by $\chi_1 \rightarrow \chi_2$, if and only if for all states S of the agent and all assignments θ, it is the case that*

$$[\chi_1]_{S,\theta,\langle v_{i_1},\dots,v_{i_k}\rangle} \subseteq [\chi_2]_{S,\theta,\langle v'_{i_1},\dots,v'_{i_k}\rangle},$$

and it is not the case that $\chi_1 \equiv \chi_2$.

In the case of implied ccc's, we execute and cache the solutions of χ_2. In order to evaluate χ_1, all we need to do is to use the cached results to restrict the solution set of χ_2.

Example 5.6. The code call condition $\mathbf{in}(\mathtt{T}_1, \mathtt{spatial}: \mathit{range}(\mathit{map}\mathtt{1}, \mathtt{5}, \mathtt{5}, \mathtt{30}))$ implies the code call condition $\mathbf{in}(\mathtt{T}_2, \mathtt{spatial}: \mathit{range}(\mathit{map}\mathtt{1}, \mathtt{5}, \mathtt{5}, \mathtt{50}))$, because all the points that are within 30 units of the point $(5, 5)$ are also within 50 units of $(5, 5)$. As mentioned above, in this case, we suppress the sequences $\langle T_1 \rangle$ and $\langle T_2 \rangle$.

Definition 5.7 (Overlapping (Sub-) CCC). Two (sub-) code call conditions χ_1 and χ_2 are said to be overlapping *wrt. the sequences $\langle v_{i_1}, \dots, v_{i_k} \rangle$ and $\langle v'_{i_1}, \dots, v'_{i_k} \rangle$, denoted by $\chi_1 \perp \chi_2$, if and only if for some states S of the agent and for some assignments θ, it is the case that*

$$[\chi_1]_{S,\theta,\langle v_{i_1},\dots,v_{i_k}\rangle} \cap [\chi_2]_{S,\theta,\langle v'_{i_1},\dots,v'_{i_k}\rangle} \neq \emptyset,$$

and neither $\chi_1 \rightarrow \chi_2$ nor $\chi_2 \rightarrow \chi_1$.

In the case of overlapping ccc's, we execute and cache the solutions of χ_3, where χ_3 is a code call condition the solution of which is set equal to the union of the solution sets of χ_1 and χ_2. In order to evaluate both χ_1 and χ_2, we need to access the cache and restrict the solution set of χ_3 to those of χ_1's and χ_2's solution sets. Note that the definition of overlapping ccc's requires that the intersection of the solution sets of χ_1 and χ_2 be non-empty for *some* state of the agent. This implies there might be states of the agent, where the intersection is empty. However, the solution set of χ_3 in such a case still contains the solutions to χ_1 and χ_2.

Example 5.8. The code call condition $\mathbf{in}(\mathtt{T}_1, \mathtt{rel}: \mathit{rngselect}(\mathit{emp}, \mathit{age}, \mathtt{25}, \mathtt{35}))$ overlaps with the code call condition $\mathbf{in}(\mathtt{T}_2, \mathtt{rel}: \mathit{rngselect}(\mathit{emp}, \mathit{age}, \mathtt{30}, \mathtt{40}))$, because all employees between the ages 30 and 35 satisfy both code call conditions.

In order to identify various relationships between code call conditions, we use the derived invariants that are computed at development phase. We are now faced with the following problem:

Definition 5.9 (Common sub-ccc identification problem). Given a set of code call conditions $\mathcal{C}$=$\{ccc_1, ccc_2, \ldots, C = ccc_n\}$, and a set of derived invariants, $\mathcal{I}^*$, find all sub-code call conditions of $\bigwedge_{i=1}^{n} ccc_i$ that are

—equivalent with respect to $\mathcal{I}^*$,

—imply one another with respect to $\mathcal{I}^*$,

—overlap with each other with respect to $\mathcal{I}^*$.

The brute-force solution to the above problem is to choose two code call conditions, ccc_i and ccc_j from $\mathcal{C}$, then traverse the list of invariants, $\mathcal{I}^*$, and apply each invariant to various sub-code call conditions of ccc_i and ccc_j. The algorithm **Brute-Force-CSI**, given in Figure 7, implements this approach.

```
Brute-Force-CSI($\mathcal{C}$,$\mathcal{I}^*$)
/* Input:   $\mathcal{C} = \{ccc_1, ccc_2, \ldots, ccc_n\}$            */
/*          $\mathcal{I}^* = \{\mathsf{inv}_1, \mathsf{inv}_2, \ldots, \mathsf{inv}_m\}$   */
/* Output:  $Eq = \{(\chi_i, \chi_j) \mid \chi_i \equiv \chi_j\}$         */
/*          $I = \{(\chi_i, \chi_j) \mid \chi_i \rightarrow \chi_j\}$          */
/*          $O = \{(\chi_i, \chi_j, \chi_k) \mid \chi_i \perp \chi_j \textit{ and } \chi_i \rightarrow \chi_k \textit{ and } \chi_j \rightarrow \chi_k\}$ /¿*/

$SC := \bigwedge_{i=1}^{n} C_i$
$SC^p$ := all sub-code call conditions of $SC$
for all $\chi_i \in SC^p$ do
   for all $\chi_j \neq \chi_i \in SC^p$ do
      for all $\mathsf{inv} \in \mathcal{I}^*$ do
         ApplyInvariant($\mathsf{inv}, \chi_i, \chi_j, Eq, I, O$)
         ApplyInvariant($\mathsf{inv}, \chi_j, \chi_i, Eq, I, O$)
Return $(Eq, I, O)$
End-Algorithm
```

Fig. 7. **Brute-Force CSI** Algorithm

The **Brute-Force-CSI** algorithm makes use of an **ApplyInvariant** routine which takes as input an invariant and two sub-code call conditions, as well as the equivalent, implied and overlapped sub-code call conditions sets. It applies the invariant to the sub-code call conditions, and inserts the relationship entailed by the invariant into the respective set. This routine is given in Figure 8. Note that we need to call **ApplyInvariant** twice with different relative orders for χ_i and χ_j.

```
ApplyInvariant($\mathsf{inv}, \chi_i, \chi_j, Eq, I, O$)

if ($iv$ is of the form $\mathsf{ic} \Longrightarrow ie_1 = ie_2$) and
   ($\exists\theta$, such that $\chi_i = (ie_1)\theta$ and $\chi_j = (ie_2)\theta$ and $(\mathsf{ic})\theta =$ true) then
     $Eq = Eq \cup \{(\chi_i, \chi_j)\}$  // $\chi_i \equiv \chi_j$
else if ($\mathsf{inv}$ is of the form $\mathsf{ic} \Longrightarrow ie_1 \subseteq ie_2$) and
        ($\exists\theta$, such that $\chi_i = (ie_1)\theta$ and $\chi_j = (ie_2)\theta$ and $(\mathsf{ic})\theta =$ true) then
          $I = I \cup \{(\chi_i, \chi_j)\}$  // $\chi_i \rightarrow \chi_j$
else if ($\mathsf{inv}$ is of the form $\mathsf{ic} \Longrightarrow (ie_1 \cup ie_2) = ie_3$) and
        ($\exists\theta$, such that $\chi_i = (ie_1)\theta$ and $\chi_j = (ie_2)\theta$ and $(\mathsf{ic})\theta =$ true) then
          $\chi_k = (ie_3)\theta$
          $O = O \cup \{(\chi_i, \chi_j, \chi_k)\}$  // $\chi_i \perp \chi_j$
Return
End-Algorithm
```

Fig. 8. **ApplyInvariant** Routine

Assuming **ApplyInvariant** takes constant time to execute, the complexity of the **Brute-Force-CSI** algorithm is $O(m * 2^{2k})$, where m is the number of invariants in $\mathcal{I}^*$ and k is the number of atomic/(in)equality code call conditions in $\mathcal{SC}$. However, one important observation is that we do not have to apply *each* invariant to *all* possible sub-code call conditions. An invariant expression can only unify with a (sub-) code call condition if both contain "similar" (sub-) code call conditions. The performance of the **Brute-Force-CSI** algorithm can be significantly improved by making use of this observation. But, before describing this improved CSI algorithm, let us first define similar (sub-) code call conditions.

Definition 5.10 (Similar (sub-) code call conditions). Two (sub-) code all conditions χ_1 and χ_2 are said to be similar *if one of the following holds:*

- *Both χ_1 and χ_2 are atomic code call conditions of the form* $\mathbf{in}(\cdot, \mathrm{d}:f(\cdot))$.
- *Both χ_1 and χ_2 are equality/inequality code call conditions.*
- *χ_1 is of the form $\chi_{11} \& \chi_{12}$ and χ_2 is of the form $\chi_{21} \& \chi_{22}$, and χ_{11} is similar to χ_{21} and χ_{12} is similar to χ_{22}.*

The **Improved-CSI** algorithm is given in Figure 9. Lines (1) and (3) of the algorithm need further explanation. In order to facilitate fast unification of sub-code call conditions with invariant expressions, the **Classify**(C) routine in the **Improved-CSI** algorithm organizes sub-code call conditions into groups such that each group contains similar sub-code call conditions. Example 5.11 demonstrates how **Classify**(C) works.

```
  Improved-CSI($C$,$\mathcal{I}^*$)
/* Input:   $\mathcal{C} = \{ccc_1, ccc_2, \ldots, ccc_n\}$
/*          $\mathcal{I}^* = \{\mathsf{inv}_1, \mathsf{inv}_2, \ldots, \mathsf{inv}_m\}$   */
/* Output:  $Eq = \{(\chi_i, \chi_j) \mid \chi_i \equiv \chi_j\}$   */
/*          $I = \{(\chi_i, \chi_j) \mid \chi_i \rightarrow \chi_j\}$   */
/*          $O = \{(\chi_i, \chi_j, \chi_k) \mid \chi_i \perp \chi_j and\ \chi_i \rightarrow \chi_k\ and\ \chi_j \rightarrow \chi_k\}$ */


(1)   $\{G_1, G_2, \ldots, G_l\}$ := Classify($C$);
(2)   for all $G_i$ for $i = 1, \ldots l$ do
(3)     $\mathcal{I} = \{\mathsf{inv} \mid \mathsf{inv}\ contains\ similar\ sub-code\ call\ conditions\ with\ G_i\}$
(4)     for all $\chi_j \in G_i$ do
(5)       for all $\chi_k \neq \chi_j \in G_i$ do
(6)         for all $\mathsf{inv} \in \mathcal{I}$ do
(7)           ApplyInvariant($\mathsf{inv}, \chi_j, \chi_k, Eq, I, O$)
(8)           ApplyInvariant($\mathsf{inv}, \chi_k, \chi_j, Eq, I, O$)
(9)   Return ($Eq, I, O$)
(10)   End-Algorithm
```

Fig. 9. **Improved CSI** Algorithm

Example 5.11. Consider the following code call conditions:

$$\begin{aligned}
\chi_{11} &= \mathbf{in}(\texttt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{sales}, \texttt{"}\geq\texttt{"}, \texttt{10K}))\\
\chi_{12} &= \texttt{FinanceRec.date} \geq \texttt{"6/6/2000"}\\
\chi_{13} &= \mathbf{in}(\texttt{C}, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \texttt{FinanceRec}, \mathit{day}))\\
\chi_{21} &= \mathbf{in}(\texttt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{sales}, \texttt{"}\geq\texttt{"}, \texttt{20K}))\\
\chi_{22} &= \texttt{FinanceRec.date} = \texttt{"7/7/2000"}\\
\chi_{23} &= \mathbf{in}(\texttt{C}, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \texttt{FinanceRec}, \mathit{day}))\\
\chi_{31} &= \mathbf{in}(\texttt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{sales}, \texttt{"}\geq\texttt{"}, \texttt{30K}))\\
\chi_{32} &= \texttt{FinanceRec.date} = \texttt{"7/7/2000"}\\
\chi_{33} &= \mathbf{in}(\texttt{C}, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \texttt{FinanceRec}, \mathit{month})).
\end{aligned}$$

Let $ccc_1 = \chi_{11}\&\chi_{12}\&\chi_{13}$, $ccc_2 = \chi_{21}\&\chi_{22}\&\chi_{23}$ and $ccc_3 = \chi_{31}\&\chi_{32}\&\chi_{33}$. Figure 10 shows how sub-code call conditions of ccc_1, ccc_2 and ccc_3 are grouped.

Line (3) of the algorithm identifies a subset $\mathcal{I} \subseteq \mathcal{I}^*$ of invariants that are applicable to a given group of sub-code call conditions. In order to speed up this task, the invariants are stored in a hash table based on the $\mathbf{in}(\cdot, \mathtt{d}: f(\cdot))$'s they contain. Given a group of sub-code call conditions, we apply only those invariants which contain similar sub-code call conditions (lines (7) and (8)). The **Improved-CSI** algorithm also uses **ApplyInvariant** to compute various relationships. However, the number of times it is invoked is much smaller than the number of times it is invoked in **Brute-Force-CSI** algorithm. Example 5.12 demonstrates how **Improved-CSI** algorithm works.

Example 5.12. The algorithm first processes group-0 (Figure 10). It identifies all invariants containing $\mathbf{in}(\cdot, \mathtt{rel}: \mathit{select}(\cdot))$ code calls. Then, it tries to apply each of those invariants to various combinations of this group. For example, the following

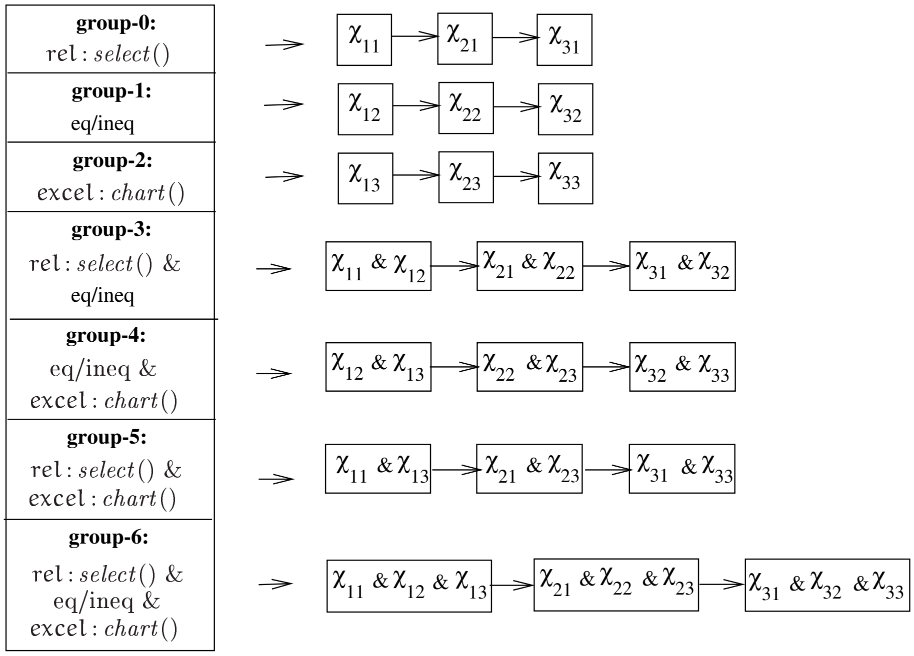

Fig. 10. Organization of Sub-code Call Conditions of Example 5.11

invariant will unify with pairs of code call conditions in this group:

$$\mathtt{Rel} = \mathtt{Rel}' \wedge \mathtt{Attr} = \mathtt{Attr}' \wedge \mathtt{Op} = \mathtt{Op}' = \texttt{"}\geq\texttt{"} \wedge \mathtt{Val} > \mathtt{Val}'$$
$$\Longrightarrow$$
$$\mathbf{in}(\mathtt{X}, \mathtt{rel}: \mathit{select}(\mathtt{Rel}, \mathtt{Attr}, \mathtt{Op}, \mathtt{V})) \subseteq \mathbf{in}(\mathtt{Y}, \mathtt{rel}: \mathit{select}(\mathtt{Rel}', \mathtt{Attr}', \mathtt{Op}', \mathtt{V}')).$$

As a result of the application of this invariant the following relationships are found:

$\mathbf{in}(\mathtt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{sales}, \texttt{"}\geq\texttt{"20K})) \rightarrow$
$\mathbf{in}(\mathtt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{sales}, \texttt{"}\geq\texttt{"}, \mathtt{10K}))$

$\mathbf{in}(\mathtt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{sales}, \texttt{"}\geq\texttt{"}, \mathtt{30K})) \rightarrow$
$\mathbf{in}(\mathtt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{sales}, \texttt{"}\geq\texttt{"}, \mathtt{10K}))$

$\mathbf{in}(\mathtt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{sales}, \texttt{"}\geq\texttt{"}, \mathtt{30K})) \rightarrow$
$\mathbf{in}(\mathtt{FinanceRec}, \mathtt{rel}: \mathit{select}(\mathit{financeRel}, \mathit{sales}, \texttt{"}\geq\texttt{"20K}))$

The same procedure is applied to group-1 resulting in the discovery of the following relationships:

$$\mathtt{FinanceRec.date} = \texttt{"7/7/2000"} \rightarrow \mathtt{FinanceRec.date} \geq \texttt{"6/6/2000"}$$
$$\mathtt{FinanceRec.date} = \texttt{"7/7/2000"} \equiv \mathtt{FinanceRec.date} = \texttt{"7/7/2000"}$$

As a result of processing group-2, the following relationship is found:

$$\mathbf{in}(\mathtt{C}, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \mathtt{FinanceRec}, \mathit{day})) \\ \equiv \\ \mathbf{in}(\mathtt{C}, \mathtt{excel}: \mathit{chart}(\mathit{excelFile}, \mathtt{FinanceRec}, \mathit{day})).$$

We process the other groups similarly. When processing group-5, we only apply invariants containing both $\mathbf{in}(\cdot, \mathtt{rel}: \mathit{select}(\cdot))$ and $\mathbf{in}(\cdot, \mathtt{excel}: \mathit{chart}(\cdot))$ code calls. Finally, the following relationships are found:

$\chi_{21} \rightarrow \chi_{11}$	$\chi_{31} \rightarrow \chi_{11}$	$\chi_{31} \rightarrow \chi_{21}$
$\chi_{22} \equiv \chi_{32}$	$\chi_{22} \rightarrow \chi_{12}$	$\chi_{32} \rightarrow \chi_{12}$
$\chi_{13} \equiv \chi_{23}$		
$\chi_{21}\&\chi_{22} \rightarrow \chi_{11}\&\chi_{12}$	$\chi_{31}\&\chi_{32} \rightarrow \chi_{11}\&\chi_{12}$	$\chi_{31}\&\chi_{32} \rightarrow \chi_{21}\&\chi_{22}$
$\chi_{22}\&\chi_{23} \rightarrow \chi_{12}\&\chi_{13}$		
$\chi_{21}\&\chi_{23} \rightarrow \chi_{11}\&\chi_{13}$		
$\chi_{21}\&\chi_{22}\&\chi_{23} \rightarrow \chi_{11}\&\chi_{12}\&\chi_{13}$		.

It is important to note that in the above algorithms the derived invariants computed during the development phase are used to match the sub-code call conditions. This assumes that the derived invariants are *complete*, that is they contain all possible relationships derivable from $\mathcal{I}$. However, this may be too costly to compute. Moreover, we may end up storing a lot of invariants which never match with any of the sub-code call conditions. One solution to this problem is to restrict the length of invariant expressions in the derived invariants. However, in that case we need to perform some inferencing at deployment to make sure that we compute all sub-code call condition relationships.

Hence, in the case of *incomplete* derived invariants, we also need to perform a second phase where we use the inference rules in Table 4 to deduce further relationships.

if $\chi_i \equiv \chi_j$ and $\chi_k \equiv \chi_l$ then $(\chi_i \& \chi_k) \equiv (\chi_j \& \chi_l)$
if $\chi_i \equiv \chi_j$ and $\chi_k \rightarrow \chi_l$ then $(\chi_i \& \chi_k) \rightarrow (\chi_j \& \chi_l)$
if $\chi_i \rightarrow \chi_j$ and $\chi_k \rightarrow \chi_l$ then $(\chi_i \& \chi_k) \rightarrow (\chi_j \& \chi_l)$

Table 4. Inference Rules Used in **Improved-CSI** Algorithm

Once the agent identifies equivalent, implied and overlapping sub-code call conditions in $\mathcal{C}$, it merges those sub-code call conditions to decrease execution costs. In the next section we will describe how to merge a set of sub-code call conditions.

5.1 Merging Code Call Conditions

In this section, we first describe a technique for evaluating costs of code call conditions. We then describe two algorithms—the **DFMerge** and the **BFMerge** algorithms—which are used to process the set $\mathcal{C} = \{ccc_1, ccc_2, ..., ccc_n\}$ of code call conditions. Both of these algorithms are parameterized by a *selection* strategy. Later, in our experiments, we will try multiple alternative selection strategies in order to determine which ones work the best. We will also compare the performance of the **DFMerge** and **BFMerge** algorithms so as to assess the efficiency of computation of these algorithms.

5.1.1 *Cost Estimation for Code Call Conditions.* In this section, we describe how to estimate the cost of merged code call conditions for a set $\mathcal{C} = \{ccc_1, ccc_2, ..., ccc_n\}$ of code call conditions. We assume that there is a cost model that can assess costs of individual code call conditions. Such costing mechanisms have been already developed for heterogeneous sources by [Du et al. 1992; Adali et al. 1996; Naacke et al. 1998; Roth et al. 1999]. Using this, we may state the cost of a single code call condition.

Definition 5.13 (Single Code Call Condition Cost). The cost of a code call condition ccc is defined as: $\boldsymbol{cost}(ccc) = \sum_{\chi_i \in ccc} \boldsymbol{cost}(\chi_i)$ *where* $\boldsymbol{cost}(\chi_i)$ *is the cost of executing the atomic or equality/inequality code call condition* χ_i*. Note that the cost of* χ_i *may include a variety of parameters such as disk/buffer read time, network retrieval time, network delays, etc.*

We may now extend this definition to describe the coalesced cost of executing two code call conditions ccc_k and ccc_{k+1}.

Definition 5.14 (Coalesced cost). The coalesced cost of executing code call conditions ccc_k *and* ccc_{k+1} *by exploiting equivalent, implied and overlapped sub-code call conditions of* ccc_k *and* ccc_{k+1} *is defined as:*

$$\boldsymbol{coalesced\_cost}(ccc_k, ccc_{k+1}) \;=\; \boldsymbol{cost}(ccc_k) + \boldsymbol{cost}(ccc_{k+1}) - \boldsymbol{gain}(ccc_k, ccc_{k+1})$$

where $\boldsymbol{gain}(ccc_k, ccc_{k+1})$ *is the cost of the savings obtained by sharing sub-code call conditions between* ccc_k *and* ccc_{k+1}.

We are now left with the problem of defining the concept of **gain** used above.

Definition 5.15 (Gain of two sub-ccc's). Suppose χ_i *and* χ_j *are sub-code call conditions in* ccc_k *and* ccc_{k+1}*, respectively, and* $\mathcal{I}$ *is a set of invariants. Then, the gain of executing* χ_i, χ_j *is defined as:*

$$\boldsymbol{gain}(\chi_i, \chi_j) = \begin{cases} \boldsymbol{cost}(\chi_i) & \text{if } \mathcal{I} \models \chi_i \equiv \chi_j \\ \boldsymbol{cost}(\chi_i) - \boldsymbol{cost}(eval(\chi_i, \chi_j)) & \text{if } \mathcal{I} \models \chi_i \rightarrow \chi_j \\ exp_k & \text{if } \mathcal{I} \models \chi_i \perp \chi_j \text{ and } \mathcal{I} \models \chi_i \rightarrow \chi_k \text{ and} \\ & \quad \mathcal{I} \models \chi_j \rightarrow \chi_k \\ 0 & \text{otherwise} \end{cases}$$

where $exp_k = \boldsymbol{cost}(\chi_i) + \boldsymbol{cost}(\chi_j) - \boldsymbol{cost}(\chi_k) - \boldsymbol{cost}(\mathrm{eval}(\chi_i, \chi_k)) - \boldsymbol{cost}(eval(\chi_j, \chi_k))$ *and* $\mathrm{eval}(\chi_i, \chi_j)$ *is the task of executing code call condition* χ_i *by using the results of code call condition* χ_j.

An explanation of the above definition is important. If code call conditions χ_i and χ_j are equivalent, then we only need to execute one of them, leading to a saving of $\mathbf{cost}(\chi_i)$. If $\chi_i \rightarrow \chi_j$ (i.e. χ_j's answers include those of χ_i) then we can first evaluate χ_j, and then select the answers of χ_i from the answers to χ_j. A third possibility is that χ_i and χ_j overlap, and there exists a code call condition χ_k such that χ_k is implied by both χ_i, χ_j. In this case, we can compute

χ_k first, and then use the result to select the answers of χ_i, χ_j. The cost of this is $\mathbf{cost}(\chi_k) + \mathbf{cost}(\mathit{eval}(\chi_i, \chi_k)) + \mathbf{cost}(\mathit{eval}(\chi_j, \chi_k))$. As the cost of executing χ_i, χ_j sequentially is $\mathbf{cost}(\chi_i) + \mathbf{cost}(\chi_j)$, the gain is computed by taking the difference, leading to the third expression. We now define the gain for two code call conditions in terms of the gains of their sub-code call conditions involved.

Definition 5.16 (Gain of two code call conditions). The gain for ccc_k and ccc_{k+1} is defined as:

$$\boldsymbol{gain}(ccc_k, ccc_{k+1}) \ = \sum_{\chi_i \in ccc_k, \chi_j \in ccc_{k+1}} \boldsymbol{gain}(\chi_i, \chi_j).$$

Example 5.17. Consider the following code call conditions:

$$\begin{aligned}
&\chi_1 : \mathbf{in}(\mathtt{FinanceRec}, \mathtt{rel} : \mathit{select}(\mathit{financeRel}, \mathit{sales}, "\geq", \mathtt{20K})),\\
&\chi_2 : \mathbf{in}(\mathtt{C}, \mathtt{excel} : \mathit{chart}(\mathtt{C}, \mathtt{FinanceRec}, \mathit{day})),\\
&\chi_3 : \mathbf{in}(\mathtt{FinanceRec}, \mathtt{rel} : \mathit{select}(\mathit{financeRel}, \mathit{sales}, "\geq", \mathtt{10K}))
\end{aligned}$$

Let $ccc_1 = \chi_1 \,\&\, \chi_2$ and $ccc_2 = \chi_3$. It is evident that $\mathrm{ANS}(\chi_1) \subseteq \mathrm{ANS}(\chi_3)$. Suppose further that the costs of these code call conditions are given as: $\mathbf{cost}\ (\chi_1) = 25$, $\mathbf{cost}\ (\chi_2) = 10$, $\mathbf{cost}\ (\chi_3) = 15$ and $\mathbf{cost}\ (\mathit{eval}\ (\chi_1, \chi_3))$=10. Then, $\mathbf{gain}\ (ccc_1, ccc_2)$ $= \sum_{\chi_i \in ccc_1, \chi_j \in ccc_2} \mathbf{gain}(\chi_i, \chi_j) = \mathbf{gain}\ (\chi_1, \chi_3)$ because $\mathbf{gain}\ (\chi_2, \chi_3)$=0, as there is no relation between code call conditions χ_2 and χ_3. As $\chi_1 \rightarrow \chi_3$, $\mathbf{gain}\ (\chi_1, \chi_3)$ = $\mathbf{cost}\ (\chi_1)$- $\mathbf{cost}\ (\mathit{eval}\ (\chi_1, \chi_3))$= 25 - 10 =15. Then, the coalesced cost of ccc_1 and ccc_2 is given by, $\mathbf{coalesced\_cost}(ccc_1, ccc_2) = \mathbf{cost}(ccc_1) + \mathbf{cost}(ccc_2) - \mathbf{gain}(ccc_1, ccc_2)$ = (25 + 10) + 15 - 15 = 35.

5.1.2 *Merging Code Call Conditions.* We now develop two algorithms that produce a *global* merged code call evaluation graph for a set, $\mathcal{C} = \{ccc_1, ccc_2, ..., ccc_n\}$ of code call conditions. These algorithms use the cceg representation of each code call condition ccc_i, and merge two graphs at a time until all graphs are merged into a final global code call evaluation graph. They make use of a **merge** routine which merges code call evaluation graphs by using the Eq, I and O sets generated by the **Improved-CSI** algorithm.

While merging two code call evaluation graphs, the **merge** routine may need to delete some nodes from the ccegs. Recall that in a cceg, a node represents an atomic/(in)equality code call condition. The following procedure is applied recursively to delete a node χ_i from a code call evaluation graph:

(1) First the node χ_i is removed.
(2) Then all incoming edges (χ_j, χ_i) and all outgoing edges (χ_i, χ_l) are deleted.
(3) If any of the nodes χ_j, encountered in the previous step, has no outgoing edges, then node χ_j is also deleted recursively.

The **merge** routine uses a set of three transformations which we define now. The first transformation takes a set of graphs of equivalent code call conditions and creates a single graph.

*Definition 5.18 (Equivalence Transformation, T1). Let $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ be of code call conditions. Let $\mathcal{CCEG} = \{cceg(C_1), cceg(C_2), \ldots, cceg(C_m)\}$ be their code call evaluation graphs. Given a set Eq of equivalent code call conditions, which are sub-cccs of the C_i's, the equivalence transformation **T1** is defined as follows:*

```
T1: $cceg(\mathcal{C}) = (\bigcup_{1 \le i \le m} V_i, \ \bigcup_{1 \le i \le m} E_i)$
    $Eq' := \{(\chi_i, \chi_j) \mid (\chi_i, \chi_j) \in Eq$ and $\nexists(\chi'_i, \chi'_j) \in Eq$ such that $\chi_i$ is a sub-ccc
                        of $\chi'_i$ and $\chi_j$ is a sub- ccc of $\chi'_j\}$

    for all $(\chi_i, \chi_j) \in Eq'$ do
        if gain$(\chi_i, \chi_j) > 0$ then
            delete all the nodes corresponding to atomic cccs in $\chi_i$
            from $cceg(\mathcal{C})$ recursively
            delete all outgoing edges $\langle \chi_i, \chi_k \rangle \in cceg(\mathcal{C})$
            create the edges $\langle \chi_j, \chi_k \rangle \in cceg\ (\mathcal{C})$
```

The second transformation **(T2)** below takes a set of graphs of code call conditions, together with a set of known implications between sub-code call conditions of these code call conditions. Using these known implications, it merges these graphs into one.

Definition 5.19 (Implication Transformation, T2). Let $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ *be of code call conditions. Let* $\mathcal{CCEG} = \{cceg(C_1), cceg(C_2), \ldots, cceg(C_m)\}$ *be their code call evaluation graphs. Given a set I of implied code call conditions, which are sub-cccs of the* C_i*'s, the implication transformation* ***T2*** *is defined as follows:*

```
T2: $cceg(\mathcal{C}) = (\bigcup_{1 \le i \le m} V_i, \ \bigcup_{1 \le i \le m} E_i)$
    $I' := \{(\chi_i, \chi_j) \mid (\chi_i, \chi_j) \in I$ and $\nexists(\chi'_i, \chi'_j) \in I$ such that $\chi_i$ is a sub-ccc
                        of $\chi'_i$ and $\chi_j$ is a sub- ccc of $\chi'_j\}$

    for all $(\chi_i, \chi_j) \in I'$ do
        if gain$(\chi_i, \chi_j) > 0$ then
            delete all incoming edges $\langle \chi_l, \chi_i \rangle \in cceg(\mathcal{C})$ to $\chi_i$ recursively
            create the edge $\langle \chi_j, \chi_i \rangle \in cceg(\mathcal{C})$
            set cost $(\chi_i)$ to cost (eval $(\chi_i, \chi_j)$)
```

The third transformation **(T3)** below takes a set of graphs of code call conditions, together with a set of known overlaps between sub-code call conditions of these code call conditions. Using these known overlaps, it merges these graphs into one.

Definition 5.20 (Overlap Transformation, T3). We consider the set $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ *of code call conditions. Let* $\mathcal{CCEG} = \{cceg(C_1), cceg(C_2), \ldots, cceg(C_m)\}$ *be their code call evaluation graphs. Given a set O of overlapping code call conditions, which are sub-cccs of* C_i*'s, the overlap transformation* ***T3*** *is defined as follows:*

```
T3: $cceg(\mathcal{C}) = (\bigcup_{1 \le i \le m} V_i, \ \bigcup_{1 \le i \le m} E_i)$
    $O' := \{(\chi_i, \chi_j, \chi_k) \mid (\chi_i, \chi_j) \in O$ and $\nexists(\chi'_i, \chi'_j, \chi'_k) \in O$ such that $\chi_i$ is a sub-ccc
                        of $\chi'_i$ and $\chi_j$ is a sub- ccc of $\chi'_j\}$

    for all $(\chi_i, \chi_j) \in O'$ do
        if gain$(\chi_i, \chi_j) > 0$ then
            create a node $\chi_k \in cceg(\mathcal{C})$
            create edges $\langle \chi_k, \chi_i \rangle \in cceg(\mathcal{C})$ and $\langle \chi_k, \chi_j \rangle \in cceg(\mathcal{C})$
            delete all incoming edges $\langle \chi_l, \chi_i \rangle \in cceg(\mathcal{C})$ to $\chi_i$ recursively
            delete all incoming edges $\langle \chi_m, \chi_j \rangle \in cceg(\mathcal{C})$ to $\chi_j$ recursively
```

```
create edges $\langle \chi_l, \chi_k \rangle \in cceg(\mathcal{C})$ and $\langle \chi_m, \chi_k \rangle \in cceg(\mathcal{C})$
set cost $(\chi_i)$ to cost (eval $(\chi_i, \chi_k)$)
set cost $(\chi_j)$ to cost (eval $(\chi_j, \chi_k)$)
```

The merge routine merely applies the above three transformations sequentially in the order **T1),(T2),(T3)**.

*Definition 5.21 (The **Merge** Routine). The **merge** routine takes as input a set of code call evaluation graphs, and the sets of equivalent, implied and overlapped sub-code call conditions, and uses **T1, T2** and **T3** to produce a single code call evaluation graph. It is given by the following:*

$$\textbf{\textit{merge}}(\mathcal{CCEG},\ Eq, I, O) = \mathbf{T3}(\mathbf{T2}(\mathbf{T1}(\mathcal{CCEG},\ Eq), I), O).$$

The merge routine works as follows: First, it gets the sets of equivalent, implied and overlapped sub-code call conditions from the **Improved-CSI** algorithm. Then, it applies the **merge**-transformations in the order: **T1, T2, T3**. The intuition behind this order is the fact that the maximum gain is obtained by merging equivalent code call conditions.

The merge routine can be utilized with any search paradigm (e.g. depth-first search, dynamic programming, etc.) to obtain an algorithm which creates a “global” code call evaluation graph. In Figures 11 and 12, we provide two algorithms that use the **merge** routine to create a global code call evaluation graph. Both algorithms merge two graphs at a time until a single graph is obtained. The **DFMerge** algorithm starts with the empty graph, and chooses the next “best” code call evaluation graph to merge with the current global code call evaluation graph. This process is iteratively executed. On the other hand, the **BFMerge** algorithm picks the “best” pair of code call evaluation graphs to merge from the ToDo list, which initially contains all code call evaluation graphs. Upon merging, the merged code call evaluation graph replaces the two code call evaluation graphs being merged. This process is executed iteratively till only one code call evaluation graph remains in the ToDo list.

```
DFMerge($\mathcal{CCEG}, Eq, I, O$)
/* Input: $\mathcal{CCEG} = \{cceg_1, ..., cceg_n\}$ */
/* Output: a global cceg                  */

$ToDo := \mathcal{CCEG}$;
$currentGraph :=$ selectNext($ToDo$, NIL);
delete $currentGraph$ from $ToDo$;
while ($ToDo$ is not empty) do
    $nextGraph :=$ selectNext($ToDo, currentGraph$);
    delete $nextGraph$ from $ToDo$;
    $currentGraph :=$ merge($\{currentGraph, nextGraph\}, Eq, I, O$);
Return $currentGraph$;
End-Algorithm
```

Fig. 11. **DFMerge** Algorithm

```
BFMerge($\mathcal{CCEG}, Eq, I, O$)
/* Input: $\mathcal{CCEG} = \{cceg_1, ..., cceg_n\}$ */
/* Output: a global cceg */

$ToDo := \mathcal{CCEG}$;
while (card($ToDo$) $> 1$ ) do
  $(cceg_i, cceg_j) :=$ selectNextPair($ToDo$);
  delete $cceg_i, cceg_j$ from $ToDo$;
  $newGraph :=$ merge($\{cceg_i, cceg_j\}, Eq, I, O$);
  insert $newGraph$ into $ToDo$;
Return $cceg_i \in ToDo$;
End-Algorithm
```

Fig. 12. **BFMerge** Algorithms

The success of both the **DFMerge** and **BFMerge** algorithms depends very much on how the next "best" merge candidates) are selected. Below, we present three alternative strategies for doing this which we have used in our experiments.

Strategy 1:

DFMerge: Choose the graph which has the largest number of equivalent code call conditions with the *currentGraph*.

BFMerge: Choose a pair of graphs which have the largest number of equivalent code call conditions.

Strategy 2:

DFMerge: Choose the graph which has the largest number of equivalent, implied or overlapped code call conditions in common with the *currentGraph*.

BFMerge: Choose a pair of graphs which have the largest number of equivalent, implied or overlapped code call conditions between the two of them.

Strategy 3:

DFMerge: Choose the graph which leads to the greatest gain with the the *currentGraph*.

BFMerge: Choose the pair of graphs the associated gain of which is maximal.

5.1.3 *Executing The Global CCEG.* The final problem that needs to be addressed is to find an execution order for the global code call evaluation graph. Any topological sort of the global cceg is a valid execution order. However, there might be several topological sorts that can be obtained from the global cceg, and some of them might be preferable to others. For example, a topological sort that gives preferences to certain nodes, i.e. outputs them earlier in the sequence, might be desirable. In order to find such an execution order, we compute weights for topological sorts.

Definition 5.22 (Weight of a topological sort). Let π be a topological sort, and $weight_{\pi(i)}$ be the weight of the ith node in the topological sort. If we have n total *nodes, the weight of π, denoted by* $weight(\pi)$*, is given by*

$$weight(\pi) = \sum_{i=1}^{n} i * weight_{\pi(i)}$$

Any topological sort that minimizes $weight(\pi)$ gives a desirable execution order. Besides, we can implement various strategies with this function simply by assigning weights accordingly. For example, if we want to favor nodes that output results, we can assign larger weights to such nodes. In order to find the topological sort with the minimum $weight(\pi)$, we use a modified topological sort algorithm which is given in Figure 13.

```
FindExecutionOrder(cceg)
/* Input: global cceg                                  */
/* Output: a topological sort $\pi$ that minimizes $weight(\pi)$ */
$D := \{v \mid v$ has indegree $0\}$
while D is not empty do
    $v'$ := node with the heighest weight in D,
    output $v'$,
    remove $v'$ from D,
    delete all outgoing edges of $v'$,
    $D := D \cup \{v \mid v$ has in-degree 0, $v \notin D\}$
End-Algorithm
```

Fig. 13. Modified Topological Sort Algorithm That Finds the Minimal $weight(\pi)$

6. EXPERIMENTS

We ran various sets of experiments on a Sun Ultra1 machine with 320 MB memory running Solaris 2.6. In the first set of these experiments, we study the execution time of the **Create-cceg** algorithm. Specifically, we evaluate the performance of this algorithm with varying number of dependencies and conjunctions in the code call conditions. In the second set of the experiments, we study the execution time of the development phase component. In particular, we study the trade-offs involved in the generic **Chk_Imp** Algorithm. In the last set of experiments, we demonstrate the efficiency of the **Improved-CSI** Algorithm, as well as the merging algorithms. We compare the performance of the merging algorithms (with different strategies) with the A^* algorithm of [Sellis 1988]. Our implementation of the development phase and deployment phase components involved over 9,500 lines of *C++* code.

6.1 Performance Evaluation of the **Create-cceg** Algorithm

To evaluate the performance of the **Create-cceg** algorithm, we generated several code call conditions, with varying number of conjuncts and number of dependencies. In the first set of experiments, we kept the number of dependencies constant and varied the number of conjuncts from 5 to 40. We repeated the same experiments when 10, 15, 20 and 25 dependencies are present. For each combination of number of dependencies and conjuncts, we created 500 code call conditions and recorded the average running time. Figure 14 shows the results. As seen from the figure, the execution time increases linearly with the number of conjuncts. The **Create-cceg** algorithm is extremely fast, taking only 14 milliseconds for code call conditions involving 40 conjunctions and 25 dependencies.

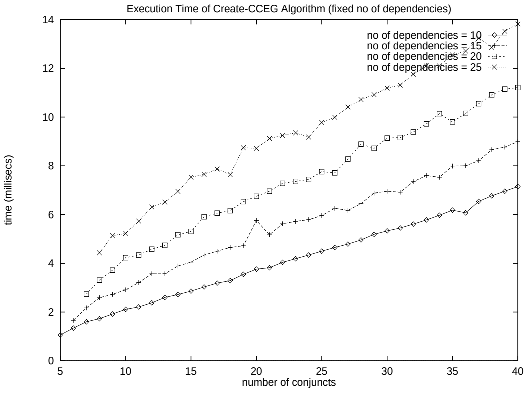

Fig. 14. Execution Time of **Create-cceg** (constant number of dependencies)

In the second, set of experiments, we kept the number of conjuncts constant, and varied the number of dependencies from 10 to 50. We ran four experiments with 10, 20, 30 and 40 number of conjuncts. Again, we generated 500 code call conditions for each combination and used the average running time. The results are given in Figure 15. Again, the execution time increases linearly with the number of dependencies.

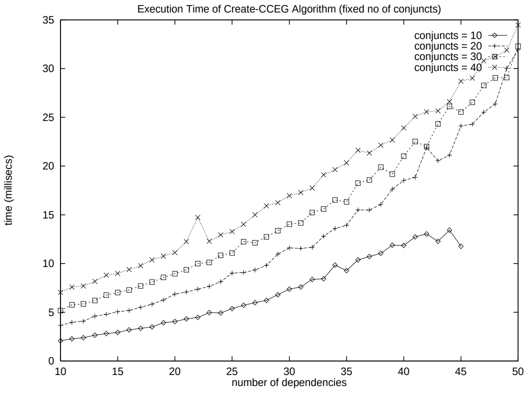

Fig. 15. Execution Time of **Create-cceg** (constant number of conjuncts)

6.2 Performance Evaluation of the Development Phase Component

In order to evaluate the performance of the development phase component, we conducted a set of experiments which use the **Chk_Ent** algorithm described in Section 4 and different instances of the **Chk_Imp** Algorithm. We varied the *threshold* and the **Axiomatic Inference System** used by **Chk_Imp**. The instances we used are described in Table 5. As the instance number increases, the complexity of the **Chk_Imp** Algorithm also increases.

Instance	**Threshold**	**Axiomatic Inference System**
Instance 0	∞	$\chi \subseteq \chi$ $\chi \cap \chi' \subseteq \chi$ $\chi \subseteq \chi \cup \chi'$
Instance i	i	All rules in Appendix B.
Instance ω	∞	All rules in Appendix B.

Table 5. Instances of the **Chk_Imp** Algorithm

We ran a set of experiments with two different data sets, namely spatial domain invariants and the relational domain invariants, which are given in Appendix C. For each instance of the algorithm we ran the development phase component several times until we get an accuracy of 3%, with 3% confidence interval. Figure 16 shows the execution time of the **Compute-Derived-Invariants** algorithm for these two data sets. As the only difference is the **Chk_Imp** Algorithm instance employed, the x-axis is labeled with those instances.

Note that the x-axis used a logarithmic-scale and hence, we may conclude that execution time increases linearly with the instance number, until instance 4096, and increases exponentially after that. However, we have observed that all instances starting from instance 4, produced the same final set of **Derived Invariants**, 18 invariants for the spatial domain, and 15 invariants for the relational domain. For the relational domain invariants, the execution-time increases more rapidly than the spatial case. The observed time increase is due to the time spent in detecting failure. Memory overflows prevented us from running experiments with larger *threshold* values.

6.3 Performance Evaluation of the Deployment Phase Component

For the performance evaluation of the deployment phase component, we ran experiments to evaluate both the execution times of the merging algorithms and the net savings obtained by the algorithms. We will describe the experimental setting in detail in the following:

In the experiments we assume a `hdb` agent that accesses relational and spatial (PR-quadtree) data sources. We have built cost estimation modules for these two sources where the cost calculations are similar to those of [Salzberg 1988]. We also built an agent cost module which coordinates with the above two modules to estimate the cost of a code call condition. The individual cost estimation modules report the cost and the cardinality of their code call conditions to the agent cost module. The agent cost model also includes network costs. For the experiments,

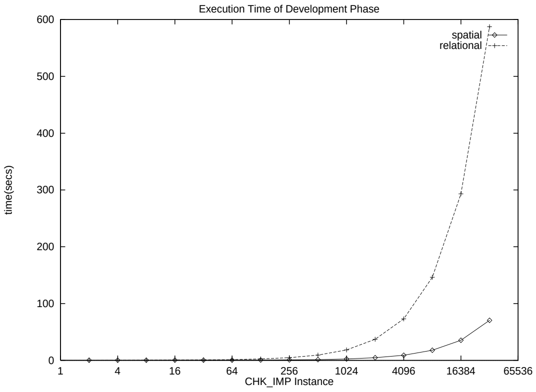

Fig. 16. Execution Time of **Compute-Derived-Invariants**

it is assumed that the data sources and the agent are on a fast Ethernet LAN. We created a synthetic database schema given below, and used the cost estimates in the experiments.

`supplier(sname, pno, quantity)`
`product(pno, price, color)`
`map(name, x-location, y-location)`
`purchases(customer_name, pno)`

We used the ccc templates given in Table 6 in the experiments. In Table 6, $Op = \{\leq, =, \geq\}$. Note that the last entry in Table 6 involves only relational data sources.

Code Call Condition Template
in(T1, rdb1 : *select2*(supplier, pno, "=", Val1, qty, Op1, Val2)) & **in**(P, quadtree : *range*(*map*, X, Y, Rad)) & =(T1.sname, P.name) & **in**(T2, rdb2 : *select*(product, price, Op2, Val3)) & =(T1.pno, T2.pno)
in(T1, rdb1 : *select2*(supplier, pno, "=", Val1, qty, Op1, Val2)) & **in**(P, quadtree : *range*(*map*, X, Y, Rad)) & =(T1.sname, P.name) & **in**(T2, rdb2 : *rngselect*(product, price, Val3, Val4)) & =(T1.pno, T2.pno)
in(T1, rdb1 : *rngselect*(supplier, qty, Val1, Val2)) & =(T1.pno, Val3) & **in**(P, quadtree : *range*(*map*, X, Y, Rad)) & =(T1.sname, P.name) & **in**(T2, rdb2 : *select*(product, price, Op2, Val3)) & =(T1.pno, T2.pno)
in(T1, rdb1 : *rngselect*(supplier, qty, Val1, Val2)) & =(T1.pno, Val3) & **in**(T2, rdb2 : *select*(product, price, Op2, Val4)) & =(T1.pno, T2.pno) & **in**(T3, rdb3 : *rngselect*(purchases, pno, Val5, Val6)) & =(T1.pno, T3.pno)

Table 6. Query Templates Used in the Experiments

By changing constants in these template code call conditions, we have created various commonality relationships. We have constructed the following three types of code call condition sets by using the above templates.

Type 1: Such sets of code call conditions only contain equivalent code call conditions.

Type 2: Such sets of code call conditions only contain both equivalent and implied code call conditions.

Type 3: Such sets of code call conditions contain equivalent,implied and overlapping code call conditions.

Before describing the experiments, let us first define the metrics we use in these sets of experiments.

Definition 6.1 (Savings Percentage). Let $\mathcal{C}$_cost be the initial total cost of the set of code call conditions, i. e., the sum of the individual code call condition costs, fin_cost be the cost of the global merged code call condition produced by the merging algorithm, IdCom_cost be the execution time of the ***Improved-CSI*** *algorithm and Merge_cost be the execution time of the merge algorithm employed. Then, the savings percentage achieved by the merge algorithm is given by:*

$$\textit{savings percentage} = \frac{\mathcal{C}\_cost - fin\_cost - IdCom\_cost - Merge\_cost}{\mathcal{C}\_cost}$$

We try to capture the net benefit of merging the code call conditions with the savings percentage metric. Moreover, in order to remedy the difference between high-cost code call conditions and low-cost code call conditions, we normalize the savings percentage metric.

Definition 6.2 (Sharing factor). Let $C = \{C_1, .., C_N\}$ be the set of given code call conditions. Let $[\chi_1], ..., [\chi_m]$ be equivalence relations, where each $[\chi_i]$ contains a set of equivalent code call conditions and $card([\chi_i]) \geq 2, i = 1, .., m$. Let $I = \{\chi_i \mid$ such that $\chi_i \notin [\chi_j]$, j=1,..,m, and there exists at least one χ_k, such that $\chi_i \rightarrow \chi_k\}$. And finally let $0 = \{(\chi_i, \chi_j, \chi_k) \mid$ such that $\chi_i, \chi_j \notin [\chi_k], k = 1, .., m,$ and $\chi_i, \chi_j \notin I,$ and $\chi_i \longleftrightarrow \chi_j, \chi_k \rightarrow \chi_i$and $\chi_k \rightarrow \chi_j\}$.

Then, the sharing factor of this set of code call conditions is given by:

$$\frac{\sum_{i=1}^{m} card([\chi_i]) * card(\chi_i) + \sum_{\chi_i \in I} card(\chi_i) + \sum_{(\chi_i,\chi_j,\chi_k) \in O} card(\chi_k)}{\sum_{i=1}^{N} \sum_{\chi_j \in C_i} card(\chi_j)}$$

The sharing factor basically gives the percentage of data objects shared among code call conditions. The intuition behind this metric is that we expect to see an increasing benefit from merging as the sharing among the code call conditions increases. In this metric, we try to avoid counting the cardinality of any code call condition more than once, so that the sharing factor is between 0% and 100%.

In order to compare our algorithms with a well known algorithm [Shim et al. 1994] for merging multiple *relational database* only queries using the A^* algorithm, we implemented an adapted version of the A^* of [Shim et al. 1994]. We used an improved version of their heuristic function. We adapted our **Improved-CSI** algorithm to work with the A^* algorithm. We enumerated the most promising 8 execution plans for each individual ccc and input those plans to the A^* algorithm.

[Sellis and Ghosh 1990] also uses similar measures. In their case, they only have equivalent relationships, hence the sharing factor metric is trivially calculated. In their version of the savings percentage metric, they only measure the difference between initial cost and the final cost obtained by merging, and fail to take into account the cost of achieving that savings. Our experiments show that although the A^* algorithm finds better global results, the cost of obtaining those results is so prohibitively high that the A^* algorithm is often infeasible to use in practice.

In all of the experiments, the algorithms are run several times to obtain results that are accurate within plus or minus 3%, with a 3% confidence interval.

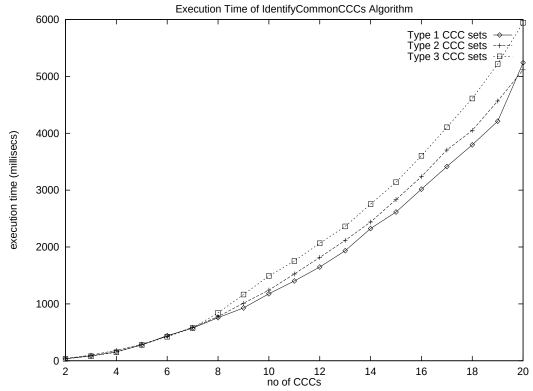

Fig. 17. Execution Time of of **Improved-CSI**

6.3.1 *The Execution Time of the **Improved-CSI** Algorithm.* The **Improved-CSI** algorithm has been ran with the three types of ccc sets. Figure 17 shows the execution times of the algorithm as the number of ccc's in the set increases. As seen from the figure, although the execution time is exponential with a small slope, it is in the order of seconds. It takes only 6 seconds for the **Improved-CSI** algorithm to find all relationships in a set containing 20 queries. Moreover, the execution time increases as more types of relationships exist in the ccc sets. It has the highest execution time for Type 3 ccc sets, and the lowest execution time for Type 1 ccc sets.

6.3.2 *Savings Achieved by the Merge Algorithms.* In these experiments, we investigate the net savings the merge algorithms achieve for our three different types of ccc sets, as well as for ccc sets involving only relational sources. We have 10 ccc's in each set. The reason for this is that the A^* algorithm exhausts memory for ccc sets having more than 11-12 ccc's.

Figures 18, 19 and 20 show the savings percentage achieved for Type 1, 2 and 3 ccc sets, respectively. As seen from Figure 18, the A^* algorithm performs as well as our merge algorithms once the sharing factor exceeds approximately 30%. We have not been able to run the A^* algorithm for low sharing factors because of the memory problem. The A^* algorithm has an effective heuristic function for equivalent ccc's, hence it is able to obtain high quality plans in a very short time. However, as seen from the figure, our merge algorithms are also able to achieve the same level of savings.

Figure 19 shows the results when there are both equivalent and implied ccc relationships. This time the heuristic function of the A^* algorithm is not as effective as with Type 1 ccc sets, and the net savings it achieves are negative until very high

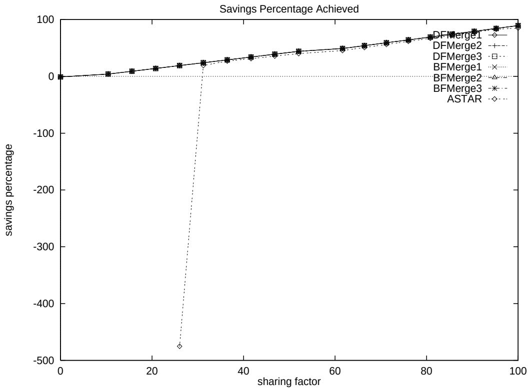

Fig. 18. Net savings achieved with Type 1 ccc Sets

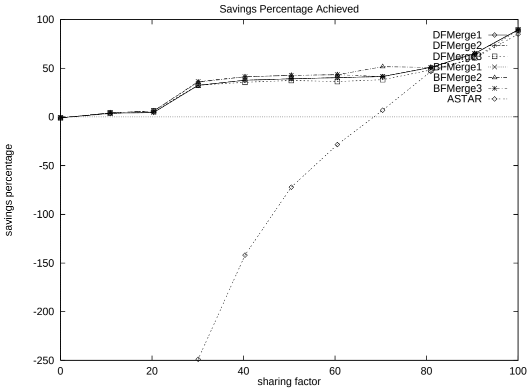

Fig. 19. Net savings achieved with Type 2 ccc Sets

sharing factors. Although the A^* algorithm finds low cost global execution plans, the execution time of the algorithm is so high that the net savings are negative. Our merge algorithms achieve very good net savings percentages. All the selection strategies perform almost equally well, with **BFMerge3** performing slightly better.

Figure 20 shows the net savings obtained when all three types of relationships exist in the ccc sets. Note that the A^* algorithm only considers equivalent and implied relationships. The results are very similar to the previous experiment. Again, our merge algorithms perform much better than the A^* algorithm. Our different select strategies have similar performances, with **BFMerge3** performing the best.

As the A^* algorithm was devised only for relational data sources, we designed another experiment involving only relational data sources. In this type of ccc sets, we only allowed equivalent relationships, as the A^* algorithm performed best with equivalent ccc's. Figure 21 shows the net savings achieved in this case. As seen from the figure, our algorithms perform as well as the A^* algorithm for sharing factor greater than 30%, and better for the rest.

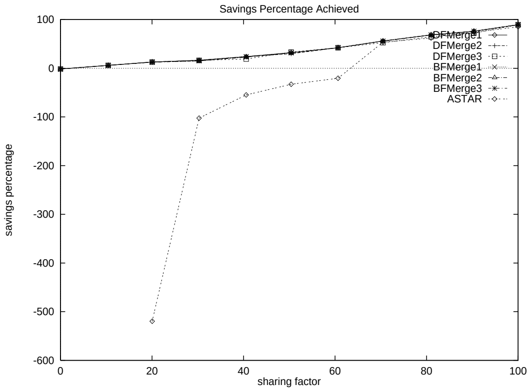

Fig. 20. Net savings achieved with Type 3 ccc Sets

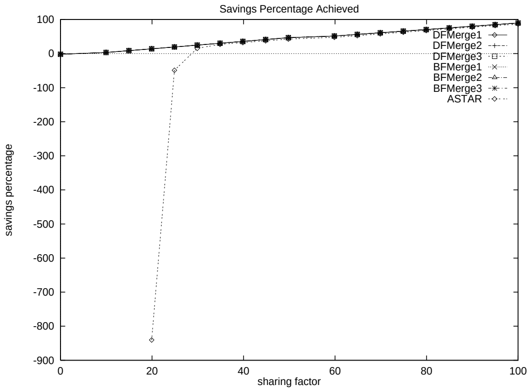

Fig. 21. Net savings achieved with relational sources

These results suggest that although our algorithms explore a smaller search space with respect to the A^* algorithm, the savings we obtain in practice are as good as that of the A^* algorithm, and the high execution cost of the A^* algorithm is prohibitive.

6.3.3 *Execution Times of Merge Algorithms.* In these experiments, we studied the execution times of our Merge algorithms and the A^* algorithm. Figures 22, 23 and 24 show the execution times for Type 1, Type 2 and Type 3 ccc sets as the number of ccc's in the sets increases. ***Note that the y-axes in the figures have logarithmic scale.*** As seen from the figures, the A^* algorithm has double-exponential execution time, and it cannot handle ccc sets having more than 10-11 ccc's, as it exhausts memory. The results show that our algorithms run (1) *1300* to *5290* times faster than the A^* algorithm for Type 1 ccc sets, (2) *1360* to *6280* times faster than the A^* algorithm for Type 2 ccc sets, and (3) *100* to *350* times faster than the A^* algorithm for Type 3 ccc sets.

The execution times of our Merge algorithms are exponential, but in the order

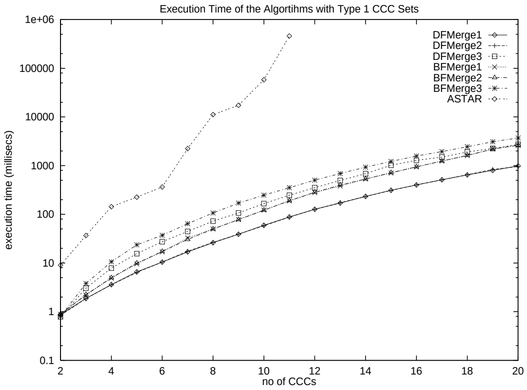

Fig. 22. Execution Time of Merge Algorithms with Type 1 ccc Sets

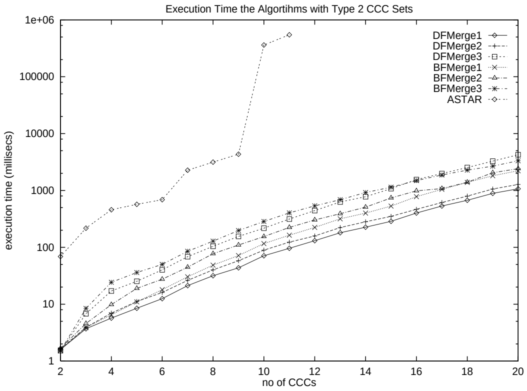

Fig. 23. Execution Time of Merge Algorithms with Type 2 ccc Sets

of milliseconds, taking less than a second for even 20 ccc's. Among our algorithms, **BFMerge3** has the highest execution time, as it uses an expensive heuristic and explores a relatively larger search space than the **DFMerge** algorithms. **DFMerge3** has the next highest execution time, and **DFMerge1** has the lowest. One important observation is that although **BFMerge3** and **DFMerge3** use a relatively expensive and more informed heuristic, and therefore have higher execution times, and find better global execution plans, they achieve the same level of net savings with the other strategies. Hence, the increased cost induced by these two strategies are not offset by the net savings they achieve.

6.3.4 *Final Cost of Plans Generated by the* ***Merge*** *Algorithms.* As the A^* algorithm examines an exhaustive search space, we studied the quality of plans generated by our **Merge** algorithms and the A^* algorithm to determine how suboptimal our final plans are. For this purpose, we examined the final costs of the plans for our Type 3 ccc sets. Figure 25 shows the estimated execution costs of the final plans generated by the **Merge** algorithms. As seen from the figure, the A^* algorithm

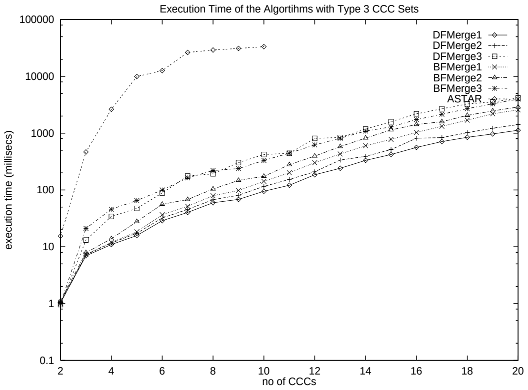

Fig. 24. Execution Time of **Merge** Algorithms with Type 3 ccc Sets

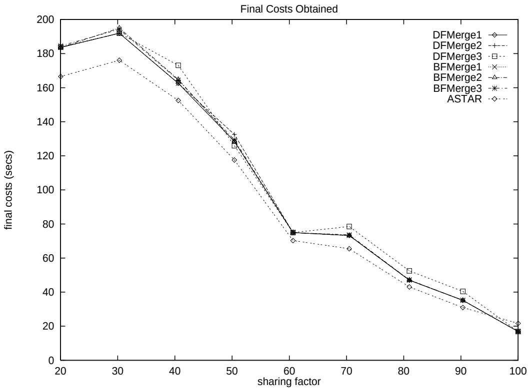

Fig. 25. Final Plan Costs Generated for Type 3 ccc Sets

almost always finds better plans than our algorithms. However, the time it spends in finding those quality plans is not offset by the net savings it achieves. Although our algorithms explore only a restricted search space, the results show that they are able to compute plans whose costs are *at most 10%* more than the plans produced by the A^* algorithm. From these results, we can conclude that our algorithms are both feasible and practical.

7. RELATED WORK

Our work has been influenced by and is related to various areas of research. Over the last decade, there has been increasing interest in building information agents that can access a set of diverse data sources. These systems include *HERMES* [Adali et al. 1996], *SchemaSQL* [Lakshmanan et al. 1996; Lakshmanan et al. 1999], *TSIMMIS* [Chawathe et al. 1994; Garcia-Molina et al. 1997], *SIMS* [Arens et al. 1993], *Information Manifold* [Levy et al. 1996b; Levy et al. 1996a], The *Internet Softbot* [Etzioni and Weld 1994], *InfoSleuth* [Bayardo et al. 1997], *Infomaster* [Genesereth et al. 1997], and *ARIADNE* [Ambite et al. 1998]. Although all these systems provide mechanisms to optimize individual requests, the only one which addresses the problem of optimizing overall agent performance is *ARIADNE*.

In [Ashish 1998; Ashish et al. 1999], the authors propose techniques to selectively materialize data to improve the performance of subsequent requests. They use the LOOM [MacGregor 1990] knowledge representation language for modeling data and maintain an ontology of classes of information sources in LOOM. They determine what to materialize by examining previous user requests as follows. They first look at the constraints imposed by user queries and create subclasses in the ontology corresponding to these restrictions. They then try to merge subclasses whenever possible. After all user queries have been examined, they sort these subclasses according to the frequency of requests and materialize subclasses from this list until the space reserved for materialization is exhausted. They repeat this process in fixed intervals. Their idea is similar to previous semantic caching ideas [Adali and Subrahmanian 1195; Adali et al. 1996; Dar et al. 1996]. In semantic caching, the cache is organized into semantic regions instead of pages. When a new query arrives, the contents of the cache is examined to determine what portion of the data requested in the query is present in the cache. A query is then created to retrieve the rest of the data from disk. The problem with semantic caching is that containment checking is hard and having a large number of semantic regions creates performance problems.

Both [Ashish et al. 1999] and [Dar et al. 1996] process one query at a time, and try to reduce the execution time by using caches, whereas we examine a set of requests (in our framework, agents can be built on top of legacy software code bases such as PowerPoint, Excel, route planners, etc. which may not support a database style query language) and try to optimize the overall execution time of this set of requests by exploiting the commonalities between them. Since we process a set of requests simultaneously, we cache the results of a code call condition evaluation only if another code call condition in this set can make use of the cached results. On the other hand, in [Ashish et al. 1999] caching decisions are based on user request histories. The advantage of their approach is that they can make use of the cache for a longer period of time, while in our case the cache contents are valid during the execution of a particular set of code call conditions. When we process the next batch of code call conditions, we discard the contents of the cache. On the other hand, the disadvantage of history based caching is that it cannot rapidly adapt to changes in interests. Nevertheless, we believe that incorporating more global level caching techniques, like the ones in [Ashish et al. 1999], into our framework is a promising research area that is worth pursuing. Another important difference is that our results also include soundness and completeness theorems.

The problem of simultaneously optimizing and merging a set of queries has been studied within the context of relational and deductive databases [Grant and Minker 1980; Sellis 1988; Shim et al. 1994; Sellis and Ghosh 1990; Finkelstein 1982; Chakravarthy and Minker 1985]. [Grant and Minker 1980; Sellis 1988; Sellis and Ghosh 1990; Shim et al. 1994] address the problem of creating a globally optimal access plan for a set of queries, provided that the common expressions among the queries are given as input. [Grant and Minker 1980] describe a branch-and-bound algorithm which searches a state space in a depth-first manner to optimize a set of relational expressions. Their algorithms are not cost-based, and hence they may increase the total execution cost of the queries. Moreover, they only consider equivalence relationships, but not containment relationships and they only deal with relational sources. Furthermore, they do not deal with non database sources.

[Sellis 1988; Shim et al. 1994; Sellis and Ghosh 1990] propose exhaustive algorithms to create a globally optimal execution plan for a set of relational database queries. [Sellis and Ghosh 1990] show that the multiple-query optimization (MQO) problem in relational databases is NP-hard even when only equivalence relationships are considered. Hence, exact algorithms for MQO are not practical and therefore, approximations or heuristic algorithms are worth pursuing.

[Sellis 1988] formulates the MQO problem as a state search problem and uses the A^* algorithm. In their approach, a *state* is defined as an n-tuple $\langle \mathrm{P}_{1j_1}, \mathrm{P}_{2j_2}, .. \mathrm{P}_{nj_n} \rangle$, where $\mathrm{P}_{1j_1} \in \{NULL\} \cup P_i$ and P_i is the set of possible access plans for query Q_i. The initial state is the vector $\langle$ NULL, …, NULL $\rangle$, that is no access plan is chosen for any query. A state transition chooses an access plan for the next query whose corresponding access plan is NULL in the state vector. The heuristic function proposed by [Sellis 1988] takes only equivalence relationships into account. [Shim et al. 1994] improves and extends this heuristic function by incorporating implication relationships and by modifying the estimated costs. This improved heuristic function provides a tighter bound than the one proposed in [Sellis 1988].

However, their approach requires enumeration of all possible plans for each query, leading to a (theoretically) very large search space. As a result, these algorithms have an exponential worst case running time. Moreover, in a heterogeneous environment, it may not be possible to assume that all query plans can be enumerated since queries might have infinitely many access plans. Furthermore, application program interfaces of individual data sources and/or software packages may not enumerate all such plans for requests shipped to them. This may be because (i) their internal code does not support it, or (ii) they are not willing to do so.

While [Sellis 1988; Shim et al. 1994; Sellis and Ghosh 1990] focus on only relational data sources, we address the problem of optimizing a set of code call conditions in agents which are built on top of arbitrary data sources. For this purpose, we provide a framework to define and identify common subexpressions for arbitrary data sources. Moreover, we do not need to enumerate all possible plans of a single query. We have implemented an adapted version of the A^* algorithm of [Shim et al. 1994] and compared it with our merging algorithms. As the results in Section 6 show, our merging algorithms are much faster than the A^*-based algorithm. As the A^*-based algorithm examines a larger search space, it may find low-cost plans that our merging algorithms may miss. However, the time it takes to find such good plans is usually not offset by the savings it achieves.

[Finkelstein 1982; Chakravarthy and Minker 1985], on the other hand, focus on detecting common expressions among a set of queries in relational and deductive databases. Since the notion of "common subexpression" varies for different data sources, the common expression identification problem for agents is very different from those of relational and deductive databases. Furthermore, they only consider equivalence and containment relationships among queries when detecting common subexpressions, whereas we also consider overlapping cases.

The only work that addresses heterogeneity and hence is most closely related to ours is that of [Subramanian and Venkataraman 1998]. The authors propose an architecture to process complex decision support queries that access to a set of heterogeneous data sources. They introduce *transient views*, which are materialized views that exist during the execution of a query. [Subramanian and Venkataraman 1998] describe algorithms which analyze the query plan generated by an optimizer to identify similar sub-plans, combine them into transient views and insert filters for compensation. Moreover, [Subramanian and Venkataraman 1998] presenst the implementation of their algorithms within the context of *DataJoiner*'s [Gupta and Lin 1994; Venkataraman and Zhang 1998] query optimizer. They try to optimize a complex decision support query by exploiting common subexpressions within this single query, whereas we try to simultaneously optimize a given set of requests. While they examine relational-style operators in detecting common subexpressions, we process any code call condition defined over arbitrary data sources not just relational sources. Moreover, they do not have a language to describe equivalence and containment relationships for heterogeneous data sources and hence these relationships are fixed apriori in the optimizer code. On the other hand, we provide invariants to describe relationships for heterogeneous data sources. Our algorithms for merging multiple code call conditions take such invariants and cost information into account when performing the merge.

Another area of research that is related to ours is partial evaluation in logic programs [Leuschel et al. 1998; Lloyd and Shepherdson 1991; De Schreye et al. 1999]. Partial evaluation takes a program and a goal and rewrites the program by using a set of transformations to optimize its performance. The rewritten program usually runs faster for the particular goal when SLD or SLD-NF resolution is used for query processing. On the other hand, our framework takes an agent program and a set of derived invariants, and tries to optimize the agent program apriori, that is at development time, prior to occurence of state changes. An interesting research problem in our framework may be the following: If a state change can be encoded as a goal, then we can use partial evaluation techniques to further optimize the rewritten agent program, as shown in Figure 26. We believe this problem needs further attention and research.

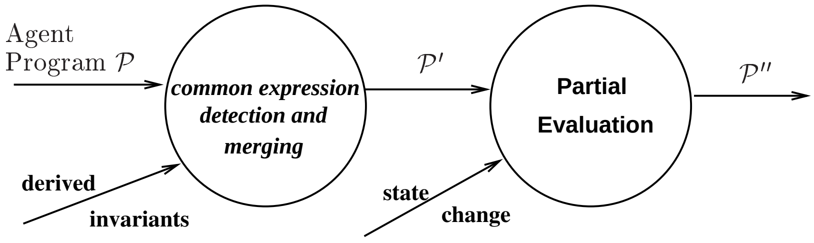

Fig. 26. Application of Partial Evaluation Techniques to Agent Programs

Another area of research that is very closely related to ours is query optimization in relational and deductive databases [Graefe 1995; Haas et al. 1989; Ioannidis and Kang 1990; Graefe 1993; Ibaraki and Kameda 1984; Kim 1982; Mumick et al. 1996], and in mediators [Adali et al. 1996; Levy et al. 1996a; Haas et al. 1997; Ambite and Knoblock 2000; Duschka et al. 2000]. It is worth noting that this list is not exhaustive since over the last decades, enormous effort has been devoted to the query optimization problem. Our work is orthogonal to techniques for optimizing individual queries, as they can be incorporated into our framework in numerous ways. For example, individual requests might be first optimized by using the techniques in [Levy et al. 1996a] or [Ambite and Knoblock 2000], then our techniques might be applied to the results. However, our focus in this paper is on the simultaneous optimization of a set of requests.

Finally, the problem of choosing appropriate materialized views to answer queries is also related to our work and there exist several papers in this area [Qian 1996; Levy et al. 1995; Chaudhuri et al. 1995]. [Levy et al. 1995] describes algorithms to determine the portions of a query that can be expressed using the definitions of materialized views. [Chaudhuri et al. 1995] identifies portions of a query that can be answered using materialized views, and determine if it is efficient to answer the query using the view. The focus of such techniques is to efficiently compute the answers to a single query, whereas our focus is to optimize the overall cost of a set of requests submitted to a heavily loaded agent.

8. CONCLUSION

There is now an incredible increase in the amount of research being conducted on software agents. Software agents now provide a host of web based services, ranging from creating personalized newspapers for people, to building multimedia presentations. In addition, agents for corporate web sites often try to personalize the web site for a given user by tracking histories of that user's interest. Agents are also being increasingly used in the aerospace and defense industries.

When an agent gets lots of requests within a short time frame, the standard mechanism that most agent frameworks use is to queue the requests in accordance with some queueing policy (e.g. LIFO, FIFO, priority queue) and then service the requests one after the other. This often leads to long wait times for requests that occur later on in the queue. In this paper, we have shown how to improve the performance of an agent by merging a set of requests and servicing the requests simultaneously. We proposed a generic and customizable framework for this purpose. Our solution applies to agents that are built on top of legacy code, which is certainly a practical assumption, as the success of the agent endeavor rests on the ability to build on top of existing data and software sources. Our solution consists of two parts:

(1) identifying "commonalities" among a set of code call conditions and

(2) computing a *single* global execution plan that simultaneously optimizes the total expected cost of this set of code call conditions.

We first provided a formal framework within which an agent developer can specify what constitutes a "common subexpression" for a data source via a set of structures, called *invariants*. *Invariants* describe *(1)* code call conditions that are "equivalent" to other code call conditions, *(2)* code call conditions that are "contained" in other code call conditions, and *(3)* code call conditions that overlap with other code call conditions. Moreover, such invariants may *imply* other invariants. We developed provably sound and complete algorithms to take the initial set of invariants input by the developer and compute *all implied* invariants.

Second, we provided an architecture to merge multiple requests in agents. We provided algorithms to identify equivalent, implied and overlapped code call conditions in any set $\mathcal{C}$. We then proposed two heuristic based algorithms, **BFMerge** and **DFMerge**, that take as input, the set of code call conditions, and produce as output, a *single* execution plan. The merging decisions are based on costs, hence the resulting global plan is guaranteed to have a reduced cost.

We have experimentally shown that our algorithms achieve significant savings. We have compared our merging algorithms with Sellis' A^*-based algorithm (which applied to merging multiple requests in the relational database case only) and demonstrated that our algorithms almost always outperform theirs. We have shown that our merging algorithms *(1)* can handle *more than twice* as many simultaneous code call conditions as the A^* algorithm and *(2)* run *100* to *6300* times faster than the A^* algorithm and *(3)* produce execution plans the cost of which is *at most 10%* more than the plans generated by the A^* algorithm.

We conclude with a brief remark on an important piece of future work. Eiter et. al. [Eiter et al. 2000] have developed a class of agents called *regular agents.* In their framework, the semantics of an agent is given by computing certain kinds of semantic constructs called "status sets." When an agent experiences a state (which may occur, for example, when it receives a message), the agent computes a new "status set" having some properties decribed in [Eiter et al. 1999]. This "status set" specifies what the agent is supposed to do in order to respond to the state change. [Subrahmanian et al. 2000] shows that this framework is rich enough not only to deal with reactive agent behavior [Kowalski and Sadri 1999], but also the so-called autonomous agent behavior of the type described by Shoham [Shoham 1993; Shoham 1999]. Eiter et. al. [Eiter et al. 2000]'s regular agent framework reduce the problem of computing "status sets" of regular agents to that of evaluating a set of code call conditions. The beauty of their result is that the syntactic restrictions on regular agents makes it possible, to associate with each agent, prior to deployment of the agent, a set of code call conditions. Whenever the agent needs to find a new status set in response to a state change, it recomputes a new status set by evaluating this set of code call conditions. Hence, all the techniques described in this paper may be used to optimize, once and for all, this set of code call conditions, so that once the agent is deployed, this optimized set of code call conditions is used by the agent for "status set" computations. We are pursuing this research avenue.

REFERENCES

Adali, S., Candan, K., Papakonstantinou, Y., and Subrahmanian, V. S. 1996. Query caching and optimization in distributed mediator systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (Montreal, Canada, June 1996), pp. 137–148.

Adali, S. and Subrahmanian, V. 1195. Intelligent caching in hybrid knowledge bases. In N. Mars Ed., *Proc. 1995 Intl. Conf. on Very Large Knowledge Bases* (Twente, The Netherlands, May 1195), pp. 247–256. IOS Press.

Ambite, J. L. et al. 1998. Ariadne: A system for constructing mediators for internet sources. In *Proceedings of ACM SIGMOD Conference on Management of Data* (Seattle, Washington, USA, June 1998), pp. 561–563.

Ambite, J. L. and Knoblock, C. A. 2000. Flexible and scalable cost-based query planning in mediators: A transformational approach. *Artificial Intelligence 118*, 1-2, 115–161.

Arens, Y., Chee, C. Y., Hsu, C.-N., and Knoblock, C. 1993. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent Cooperative Information Systems 2*, 2, 127–158.

Ashish, N. 1998. Optimizing information agents by selectively materializing data. In *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference* (Madison, Wisconsin, USA, July 1998), pp. 1168. Doctoral Consortium Abstract.

Ashish, N., Knoblock, C. A., and Shahabi, C. 1999. Selectively materializing data in mediators by analyzing user queries. In *Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems, (CoopIS)* (Edinburgh, Scotland, September 1999), pp. 256–266.

Bayardo, R. et al. 1997. Infosleuth: Agent-based semantic integration of information in open and dynamic environments. In J. Peckham Ed., *Proceedings of ACM SIGMOD Conference on Management of Data* (Tucson, Arizona, 1997), pp. 195–206.

Calvanese, D., De Giacomo, G., and Lenzerini, M. 1998. On the decidability of query containment under constraints. In *Proc. of the 17th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)* (Seattle, Washington, USA, June 1998), pp. 214–223.

Chakravarthy, U. S. and Minker, J. 1985. Multiple query processing in deductive databases using query graphs. In *Proc. of the Conf. on Very Large Data Bases (VLDB)* (Kyoto, Japan, 1985), pp. 384–391.

Chaudhuri, S., Krishnamurthy, R., Potamianos, S., and Shim, K. 1995. Optimizing queries with materialized views. In *Proceedings of IEEE Conference on Data Engineering* (Taipei, Taiwan, March 1995), pp. 190–200.

Chawathe, S. et al. 1994. The tsimmis project: Integration of heterogeneous information sources. In *Proceedings of the 10th Meeting of the Information Processing Society of Japan* (Tokyo, Japan, October 1994).

Dar, S., Franklin, M. J., Jonsson, B., Srivastava, D., and Tan, M. 1996. Semantic data caching and replacement. In *Proc. of the Conf. on Very Large Data Bases (VLDB)* (Bombay, India, September 1996), pp. 330–341.

De Schreye, D. et al. 1999. Conjunctive partial deduction: foundations, control, algorithms and experiments. *Journal of Logic Programming 41*, 2-3, 231–277.

Dix, J., Kraus, S., and Subrahmanian, V. 2001. Temporal agent reasoning. *Artificial Intelligence to appear.*

Dix, J., Nanni, M., and Subrahmanian, V. S. 2000. Probabilistic agent reasoning. *Transactions of Computational Logic 1*, 2, 201–245.

Dix, J., Subrahmanian, V. S., and Pick, G. 2000. Meta Agent Programs. *Journal of Logic Programming 46*, 1-2, 1–60.

Du, W., Krishnamurthy, R., and Shan, M.-C. 1992. Query optimization in heterogeneous DBMS. In *Proc. of the Conf. on Very Large Data Bases (VLDB)* (Vancouver, Canada, August 1992), pp. 277–291.

Duschka, O. M., Genesereth, M. R., and Levy, A. Y. 2000. Recursive query plans for data integration. *Journal of Logic Programming 43*, 1 (April), 49–73.

Eiter, T., Subrahmanian, V., and Pick, G. 1999. Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence 108*, 1-2, 179–255.

Eiter, T., Subrahmanian, V., and Rogers, T. J. 2000. Heterogeneous Active Agents, III: Polynomially Implementable Agents. *Artificial Intelligence 117*, 1, 107–167.

Etzioni, O. and Weld, D. 1994. A softbot-based interface to the internet. *Communications of the ACM 37*, 7, 72–76.

Finkelstein, S. 1982. Common expression analysis in database applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data* (Orlando, Florida, USA, 1982).

Garcia-Molina, H. et al. 1997. The tsimmis approach to mediation: Data models and languages. *Journal of Intelligent Information Systems 8*, 2, 117–132.

GENESERETH, M. R., KELLER, A. M., AND DUSCHKA, O. M. 1997. Infomaster: An information integration system. In *Proceedings of ACM SIGMOD Conference on Management of Data* (Tucson, Arizona, USA, May 1997), pp. 539–542.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys 25*, 2, 73–170.

GRAEFE, G. 1995. The cascades framework for query optimization. *Bulletin of the TC on Data Engineering 18*, 3 (September), 19–29.

GRANT, J. AND MINKER, J. 1980. On optimizing the evaluations of a set of expressions. Technical Report TR-916 (July), Dept. of Computer Science, Univ. of Maryland, College Park, MD, USA.

GUPTA, P. AND LIN, E. T. 1994. Datajoiner: A practical approach to multi-database access. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems* (Austin, Texas, September 1994). IEEE-CS Press.

HAAS, L. M., FREYTAG, J. C., LOHMAN, G. M., AND PIRAHESH, H. 1989. Extensible query processing in starburst. In *Proceedings of ACM SIGMOD Conference on Management of Data* (Portland, OR, USA, 1989), pp. 377–388.

HAAS, L. M., KOSSMANN, D., WIMMERS, E. L., AND YANG, J. 1997. Optimizing queries across diverse data sources. In *Proceedings of the 23rd Int. Conference on Very Large Databases (VLDB)* (Athens, Greece, August 1997), pp. 276–285.

IBARAKI, T. AND KAMEDA, T. 1984. On the optimal nesting order of computing n-relational joins. *ACM Transactions on Database Systems 9*, 3 (September), 482–502.

IOANNIDIS, Y. E. AND KANG, Y. C. 1990. Randomized algorithms for optimizing large join queries. In *Proceedings of ACM SIGMOD Conference on Management of Data* (Atlantic City, NJ, USA, May 1990), pp. 312–321.

KIM, W. 1982. On optimizing an sql-like nested query. *ACM Transactions on Database Systems 7*, 3 (September), 443–469.

KNUTH, D. E. 1997. *The Art of Computer Programming, Volume I: Fundamental Algorithms* (3rd ed.). Addison-Wesley.

KOWALSKI, R. AND SADRI, F. 1999. From logic programming to multi-agent systems. *Annals of Mathematics and Artificial Intelligence, Special issue edited by Jürgen Dix and Jorge Lobo, 25*, 3-4, 391–419.

LAKSHMANAN, L. V. S., SADRI, F., AND SUBRAMANIAN, I. N. 1996. Schemasql - a language for interoperability in relational multi-database systems. In *Proceedings of the 22nd Int. Conference on Very Large Databases, (VLDB)* (Bombay, India, September 1996), pp. 239–250.

LAKSHMANAN, L. V. S., SADRI, F., AND SUBRAMANIAN, S. N. 1999. On efficiently implementing schemasql on a sql database system. In *Proceedings of the 25th Int. Conference on Very Large Databases, (VLDB)* (Edinburgh, Scotland, September 1999), pp. 471–482.

LEUSCHEL, M., MARTENS, D., AND DE SCHREYE, D. 1998. Some achievements and prospects in partial deduction. *ACM Computing Surveys 30*, 3es (September).

LEVY, A., RAJARAMAN, A., AND ORDILLE, J. 1996a. Querying heterogeneous information sources using source descriptions. In *Proceeding of the 22nd Int. Conference on Very Large Databases (VLDB)* (Bombay, India, September 1996), pp. 251–262.

LEVY, A. Y., MENDELZON, A. O., SAGIV, Y., AND SRIVASTAVA, D. 1995. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)* (San Jose, CA, USA, May 1995), pp. 95–104.

LEVY, A. Y., RAJARAMAN, A., AND ORDILLE, J. J. 1996b. The world wide web as a collection of views: Query processing in the information manifold. In *Workshop on Materialized Views: Techniques and Applications (VIEW 1996)* (Montreal, Canada, 1996), pp. 43–55.

LLOYD, J. W. AND SHEPHERDSON, J. C. 1991. Partial evaluation in logic programming. *Journal of Logic Programming 11*, 3-4, 217–242.

MACGREGOR, R. 1990. The evolving technology of classification-based knowledge representation systems. In J. SOWA Ed., *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. San Mateo, CA: Morgan Kaufmann.

MUMICK, I. S., FINKELSTEIN, S. J., PIRAHESH, H., AND RAMAKRISHNAN, R. 1996. Magic conditions. *ACM Transactions on Database Systems 21*, 1 (March), 107–155.

NAACKE, H., GARDARIN, G., AND TOMASIC, A. 1998. Leveraging mediator cost models with heterogeneous data sources. In *Proc. IEEE Conf. on Data Engineering* (Orlando, Florida, USA, 1998), pp. 351–360.

QIAN, X. 1996. Query folding. In *Proc. IEEE Conf. on Data Engineering* (New Orleans, LA, USA, 1996), pp. 48–55.

ROTH, M. T., OZCAN, F., AND HAAS, L. 1999. Cost models do matter: Providing cost information for diverse data sources in a federated system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)* (Edinburgh, Scotland, UK, September 1999).

SALZBERG, B. 1988. *File Structures: An Analytical Approach.* Prentice Hall.

SAMET, H. 1989. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA.

SELLIS, T. K. 1988. Multiple query optimization. *ACM Transactions on Database Systems 13*, 1 (March), 23–52.

SELLIS, T. K. AND GHOSH, S. 1990. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering 2*, 2 (June), 262–266.

SHIM, K., SELLIS, T. K., AND NAU, D. 1994. Improvements on a heuristic algorithm for multiple-query optimization. *Data and Knowledge Engineering 12*, 2, 197–222.

SHOHAM, Y. 1993. Agent Oriented Programming. *Artificial Intelligence 60*, 51–92.

SHOHAM, Y. 1999. What we talk about when we talk about software agents. *IEEE Intelligent Systems 14*, 28–31.

SUBRAHMANIAN, V., BONATTI, P., DIX, J., EITER, T., KRAUS, S., OZCAN, F., AND ROSS, R. 2000. *Heterogeneous Agent Systems.* MIT Press, Cambridge, MA.

SUBRAMANIAN, S. N. AND VENKATARAMAN, S. 1998. Cost-based optimization of decision support queries using transient views. In *Proceedings of ACM SIGMOD Conference on Management of Data* (Seattle, WA, USA, June 1998), pp. 319–330.

ULLMAN, J. D. 1989. *Principles of Database and Knowledge Base Systems*, Volume 2. Computer Science Press, New York, NY.

VENKATARAMAN, S. AND ZHANG, T. 1998. Heterogeneous database query optimization in db2 universal datajoiner. In *Proceedings of the 24th Int. Conference on Very Large Databases, (VLDB)* (New York City, USA, August 1998), pp. 685–689.

APPENDIX

A. PROOFS OF THEOREMS

PROOF OF THEOREM 2.10.

($\Rightarrow$): Suppose π is a witness to the safety of χ. There are two cases:

Case 1: Let $\chi_{\pi(i)}$ be an atomic code call condition of the form $\mathbf{in}(\mathtt{X}_{\pi(\mathtt{i})}, \mathtt{cc}_{\pi(\mathtt{i})})$, then by the definition of safety, $root(\mathtt{cc}_{\pi(\mathtt{i})}) \subseteq RV_\pi(i)$, where $RV_\pi(i) = \{root(\mathtt{Y}) \mid \exists j < i \; s.t. \; Y \; occurs \; in \; \chi_{\pi(j)}\}$, and either $\mathtt{X}_{\pi(\mathtt{i})}$ is a root variable or $root(\mathtt{X}_{\pi(\mathtt{i})}) \in RV_\pi(i)$. Then, there exist $\chi_{\pi(j_1)}, \chi_{\pi(j_2)}, \ldots, \chi_{\pi(j_k)}, j_k <$ i , such that $root(\mathtt{X}_{\pi(\mathtt{j_k})}) \subseteq RV_\pi(i)$, and $root(\mathtt{X}_{\pi(\mathtt{j_k})}) \subseteq root(\mathtt{cc}_{\pi(\mathtt{i})})$. But, then $\chi_{\pi(i)}$ is dependent on each of the $\chi_{\pi(j_1)}, \chi_{\pi(j_2)}, \ldots, \chi_{\pi(j_k)}, j_k <$ i by definition. Hence, there exist edges

$$(\chi_{\pi(j_1)}, \chi_{\pi(i)}), (\chi_{\pi(j_2)}, \chi_{\pi(i)}), \ldots (\chi_{\pi(j_k)}, \chi_{\pi(i)}).$$

Therefore, $\chi_{\pi(j_1)}, \chi_{\pi(j_2)}, \ldots, \chi_{\pi(j_k)}, j_k <$ i precede $\chi_{\pi(i)}$, hence π is also a topological sort of the cceg of χ.

Case 2: If $\chi_{\pi(i)}$ is an equality/inequality of the form $\mathtt{s}_1$ op $\mathtt{s}_2$, then at least one of $\mathtt{s}_1$, $\mathtt{s}_2$ is a constant or a variable $\mathtt{S}$ such that $\mathit{root}(\mathtt{S}) \in RV_{\pi(i)}$. Suppose at least one of $\mathtt{s}_1$, $\mathtt{s}_2$ is a variable. Then, there exists a $\chi_{\pi(j)}$, j < i, such that $\mathit{root}(\mathtt{S}) \in \mathit{root}(\mathtt{X}_{\pi(\mathtt{j})})$, as $\mathit{root}(X_{\pi(j)}) \subseteq RV_\pi(i)$. But, then $\chi_{\pi(i)}$ is dependent on $\chi_{\pi(j)}$ by definition, and there exists an edge $(\chi_{\pi(j)}, \chi_{\pi(i)})$ in the cceg of χ. Hence, $\chi_{\pi(j)}$ precedes $\chi_{\pi(i)}$ in the topological sort of the cceg. If both $\mathtt{s}_1$ and $\mathtt{s}_2$ are constants, then their nodes have in-degree 0 in the cceg, and no code call condition needs to precede $\chi_{\pi(i)}$ in the topological sort order, i.e., they are unrestricted. Therefore, π is also a topological sort of the cceg of χ.

($\Leftarrow$): Suppose π is a topological sort of the cceg of χ. Let

$$\chi_{\pi(i)}, \chi_{\pi(j_1)}, \chi_{\pi(j_2)}, \ldots, \chi_{\pi(j_k)},\ j_k < i$$

be code call conditions such that there exist edges

$$(\chi_{\pi(j_1)}, \chi_{\pi(i)}), (\chi_{\pi(j_2)}, \chi_{\pi(i)}), \ldots, (\chi_{\pi(j_k)}, \chi_{\pi(i)})$$

in the cceg of χ. Then, by definition each $\chi_{\pi(j_m)}$, $m = 1, \ldots, k$, depends on $\chi_{\pi(i)}$. If $\chi_{\pi(i)}$ is an atomic code call condition of the form $\mathbf{in}(\mathtt{X}_{\pi(\mathtt{i})}, \mathtt{cc}_{\pi(\mathtt{i})})$, then $\mathit{root}(\mathtt{X}_{\pi(\mathtt{j_m})}) \subseteq \mathit{root}(\mathtt{cc}_{\pi(\mathtt{i})}), m = 1, \ldots, k$. As $\forall j_m, m = 1, \ldots, k, j_m < i$, $\mathit{root}(\mathtt{X}_{\pi(\mathtt{j_m})}) \subseteq RV_{\pi(i)}$, by definition of $RV_{\pi(i)}$, hence $\mathit{root}(\mathtt{cc}_{\pi(\mathtt{i})}) \subseteq RV_{\pi(i)}$. On the other hand, if $\chi_{\pi(i)}$ is an equality/inequality of the form $\mathtt{s}_1$ op $\mathtt{s}_2$, then either $\mathtt{s}_1$ is a variable and $\mathit{root}(\mathtt{s}_1) \in \mathit{root}(\mathtt{X}_{\pi(\mathtt{j_m})})$, where $j_m \in \{j_1, \ldots, j_k\}$, or $\mathtt{s}_2$ is a variable and $\mathit{root}(\mathtt{s}_1) \in \mathit{root}(\mathtt{X}_{\pi(\mathtt{j'_m})})$, where $j'_m \in \{j_1, \ldots, j_k\}$, or both. But, $\mathit{root}(\mathtt{X}_{\pi(\mathtt{j_m})}) \subseteq RV_{\pi(i)}\ \forall j_m, m = 1, \ldots, k, j_m < i$. Hence, $\mathit{root}(\mathtt{s}_1), \mathit{root}(\mathtt{s}_2) \in \mathit{root}(\mathtt{X}_{\pi(\mathtt{j_m})})$. If both $\mathtt{s}_1$ and $\mathtt{s}_2$ are constants, then they are unrestricted in the topological sort. Therefore, π is also a witness to the safety of χ. ☐

PROOF OF THEOREM 2.20. The proof is by induction on the structure of condition lists.

Base Cases: Base cases are when the condition list consists of t_1 Op t_2 where $\mathsf{Op} \in \{<, >, \leq, \geq, =\}$ and each of t_1, t_2 is either a variable or a constant. We suppress the cases when both t_1, t_2 are constants: the relation either holds (in that case we an eliminate t_1 Op t_2) or it does not (in that case we can eliminate the whole invariant).

Op = "$\leq, \geq$": We have to consider terms of the form $t_1 \leq t_2$ (resp. $t_1 \geq t_2$) and distinguish the following cases. For each case we define expressions $\mathsf{ie}'_1, \mathsf{ie}'_2$ such that true $\Longrightarrow\ \mathsf{ie}'_1\ \Re\ \mathsf{ie}'_2$ is equivalent to $t_1 \leq t_2 \Longrightarrow\ \mathsf{ie}_1\ \Re\ \mathsf{ie}_2$.

(1) t_2 *is a constant* a*:* Then t_1 is a variable. We modify $\mathsf{ie}_1, \mathsf{ie}_2$ by introducing a new variable $\mathtt{X_{new}}$ and adding the following ccc to all subexpressions of $\mathsf{ie}_1, \mathsf{ie}_2$ containing t_1

$$\mathbf{in}(\mathtt{t}_1, \mathtt{ag} : \mathit{subtraction}(\mathtt{a}, \mathtt{X_{new}}))\ \&\ \mathbf{in}(\mathtt{1}, \mathtt{ag} : \mathit{geq\_0}(\mathtt{X_{new}})).$$

We note that t_1 now becomes an auxiliary variable and $\mathtt{X_{new}}$ is a base variable.
$\mathfrak{Trans}(\mathsf{ic}, \mathsf{ie}_i)$ is defined to be the modified ie_i just described.

(2) t_1 *is a constant* a*:* Then t_2 is a variable. We modify $\mathsf{ie}_1, \mathsf{ie}_2$ by introducing a new variable $\mathtt{X_{new}}$ and adding the following ccc to all subexpressions of $\mathsf{ie}_1, \mathsf{ie}_2$ containing t_2

$$\mathbf{in}(\mathtt{t}_2, \mathfrak{ag} : \mathit{addition}(\mathsf{a}, \mathtt{X_{new}})) \ \& \ \mathbf{in}(\mathtt{1}, \mathfrak{ag} : \mathit{geq\_0}(\mathtt{X_{new}})).$$

Again, t_2 becomes an auxiliary variable and $\mathtt{X_{new}}$ is a base variable. $\mathfrak{Trans}(\mathsf{ic}, \mathsf{ie}_i)$ is defined to be the modified ie_i just described.

(3) *Both* t_1, t_2 *are variables:* We modify $\mathsf{ie}_1, \mathsf{ie}_2$ by introducing a new variable $\mathtt{X_{new}}$ and adding the following ccc to all subexpressions of $\mathsf{ie}_1, \mathsf{ie}_2$ containing t_2

$$\mathbf{in}(\mathtt{t}_2, \mathfrak{ag} : \mathit{addition}(\mathtt{t}_1, \mathtt{X_{new}})) \ \& \ \mathbf{in}(\mathtt{1}, \mathfrak{ag} : \mathit{geq\_0}(\mathtt{X_{new}})).$$

Again, t_2 becomes an auxiliary variable and $\mathtt{X_{new}}$ is a base variable. $\mathfrak{Trans}(\mathsf{ic}, \mathsf{ie}_i)$ is defined to be the modified ie_i just described.

The case $\geq$ is completely analogous: just switch t_1 with t_2. Note that the above covers all possible cases, as any variable in the condition list must be a base variable (see Definition 2.17).

Op = **"¡,¿":** Analogous to the previous case, just replace "$\mathfrak{ag} : \mathit{geq\_0}(\mathtt{X_{new}})$" by "$\mathfrak{ag} : \mathit{ge}_{\mathit{0}}(\mathtt{X_{new}})$"

Op = **"=":** If in $t_1 = t_2$ the term t_1 is a variable, then we replace each occurrence of t_1 in $\mathsf{ie}_1, \mathsf{ie}_2$ by t_2. If t_1 is a constant and t_2 is a variable, replace each occurrence of t_2 in $\mathsf{ie}_1, \mathsf{ie}_2$ by t_1.

Inductive Step: As the condition list is just a conjunction of the cases mentioned above, we can apply our modifications of $\mathsf{ie}_1, \mathsf{ie}_2$ one after another. Once all modifications have been performed, we arrive at an equivalent formula of the form

$$\text{true} \Longrightarrow \ \mathfrak{Trans}(\mathsf{ic}, \mathsf{ie}_1) \ \Re \ \mathfrak{Trans}(\mathsf{ic}, \mathsf{ie}_2) \quad \square$$

PROOF OF COROLLARY 2.21.
($\Rightarrow$) : Let $\mathsf{inv} : \mathsf{ic} \Longrightarrow \ \mathsf{ie}_1 \ \Re \ \mathsf{ie}_2$ be an invariant. We can assume that ic is in DNF: $C_1 \vee C_2 \vee \ldots \vee C_m$. Thus we can write inv as follows:

$$\{C_i \Longrightarrow \ \mathsf{ie}_1 \ \Re \ \mathsf{ie}_2 \mid 1 \leq i \leq m\}$$

Let $\mathsf{inv}\theta$ be any ground instance of inv. If $(S, \theta) \models \mathsf{inv}$, then either $(C_1 \vee C_2 \vee \ldots \vee C_m)\theta$ evaluates to false in state S, or $(\mathsf{ie}_1)\theta \ \Re \ (\mathsf{ie}_2)\theta$ is true in S. Assume that $(C_1 \vee C_2 \vee \ldots \vee C_m)\theta$ evaluates to false, then each $(C_i)\theta$ has to be false in S. Hence,

$$(S, \theta) \models (C_i \Longrightarrow \ \mathsf{ie}_1 \ \Re \ \mathsf{ie}_2) for 1 \leq i \leq m$$

Assume $(C_1 \vee C_2 \vee \ldots \vee C_m)\theta$ evaluates to true in S. Then there exists at least one $(C_i)\theta$ that evaluates to true in state S. Let $T = \{(C_j)\theta \mid 1 \leq j \leq m\}$ be the set of conjunctions that are true in S. As all other $(C_i)\theta \ \notin T$ evaluates to false, $(S, \theta) \models (C_i \Longrightarrow \mathsf{ie}_1 \Re \, \mathsf{ie}_2)$ *for* $1 \leq i \leq m$, *and* $(C_i)\theta \ \notin T$. But $(S, \theta) \models \mathsf{inv}$, hence $(\mathsf{ie}_1)\theta \ \Re \ (\mathsf{ie}_2)\theta$ is true in S. As a result, $(S, \theta) \models (C_j \Longrightarrow \ \mathsf{ie}_1 \ \Re \ \mathsf{ie}_2)$ *for* $1 \leq j \leq m$ *and*$(C_j)\theta \in T$.

Since, each $C_i \Longrightarrow \ \mathsf{ie}_1 \Re \, \mathsf{ie}_2$ is an ordinary invariant the result follows from Theorem 2.20.

($\Leftarrow$) : Assume that $(\forall C_i, 1 \leq i \leq m)$ $(S,\theta) \models \text{true} \Longrightarrow \mathfrak{Trans}(C_i, \mathsf{ie}_1)\ \Re\ \mathfrak{Trans}(C_i, \mathsf{ie}_2)$ and suppose $(S,\theta) \models (C_1 \vee C_2 \vee \ldots \vee C_m)$. Then by Theorem 2.20, $(\forall C_i, 1 \leq i \leq m)$ $(S,\theta) \models (C_i \Longrightarrow \mathsf{ie}_1\ \Re\ \mathsf{ie}_2)$. There exists at least one $(C_j)\theta$ which evaluates to true in S. But then, $(\mathsf{ie}_1)\theta\ \ \Re\ \ (\mathsf{ie}_2)\theta$ is true in state S. Hence, $(S,\theta) \models \mathsf{inv}$. ☐

Proof of Lemma 4.4.

$$\begin{gathered}\textbf{Chk\_Imp}(\mathsf{ie}_1, \mathsf{ie}_2)\\ \textit{if and only if}\\ \text{for all states } S \text{ and all assignments } \theta\text{: } [\mathsf{ie}_1]_{S,\theta} \subseteq [\mathsf{ie}_2]_{S,\theta}\\ \textit{if and only if}\\ \text{for all states } S \text{ and all assignments } \theta\text{: true} \Longrightarrow \mathsf{ie}_1\theta \subseteq \mathsf{ie}_2\theta\\ \textit{if and only if}\\ \textbf{Chk\_Taut}(\text{true} \Longrightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}_2).\end{gathered}$$

(2) follows from Theorem 2.20 and Corollary 2.21. Note that it also holds for invariants of the form $\mathsf{ic} \Rightarrow \mathsf{ie}_1 = \mathsf{ie}_2$ because they can be written as two separate invariants: "$\mathsf{ic} \Rightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}_2$" and "$\mathsf{ic} \Rightarrow \mathsf{ie}_1 \supseteq \mathsf{ie}_2$". *(3)* is immediate by the very definition. ☐

Proof of Proposition 4.5. We show that the containment problem [Ullman 1989] in the relational model of data is an instance of the problem of checking implication between invariant expressions. The results follow then from Lemma 4.4 and the fact, that the containment problem in relational databases is well known to be undecidable.

To be more precise, we use the results in [Calvanese et al. 1998], where it has been shown that in the relational model of data, the *containment of conjunctive queries containing inequalities* is undecidable. It remains to show that our implication check problem between invariant expressions can be reduced to this problem.

Let $\mathtt{relational}:\mathit{query}(Q)$ be a code call that takes as input an arbitrary set of subgoals corresponding to the conjunctive query Q and returns as output the result of executing Q.

Let Q_1 and Q_2 be arbitrary conjunctive queries which may contain inequalities. we define

$$\mathsf{ie}_1 = \mathtt{relational}:\mathit{query}(Q_1), \quad \mathsf{ie}_2 = \mathtt{relational}:\mathit{query}(Q_2).$$

Then, clearly

$$\textbf{Chk\_Imp}(\mathsf{ie}_1, \mathsf{ie}_2) = \textbf{true} \ \ \textit{if and only if} \ \ Q_1 \subseteq Q_2.$$

Hence the implication check problem is also undecidable. ☐

Proof of Proposition 4.6. Clearly, by Lemma 4.4, it suffices to prove the proposition for **Chk_Imp**.

For an invariant expression ie, the set of all substitutions θ such that $\mathsf{ie}\theta$ is ground, is finite (because of our assumption about finiteness of the domains of all datatypes). Thus, our atomic code call conditions $\mathbf{in}(\mathtt{obj}, \mathtt{ag}:f(\mathtt{args}))$ can all be seen as propositional variables. Therefore, using this restriction, we can view our formulae as *propositional* formulae and a *state* corresponds to a *propositional valuation*.

With this restriction, our problem is certainly in co-NP, because computing $[\mathsf{ie}]_{S,\theta}$ is nothing but evaluating a propositional formula (the valuation corresponds to the state S). Thus "$[\mathsf{ie}_1]_{S,\theta} \subseteq [\mathsf{ie}_2]_{S,\theta}$ for all S and all assignments θ" translates to checking whether a propositional formula is a tautology: a problem known to be in co-NP.

To show completeness, we use the fact that checking whether C is a logical consequence of $\{C_2, \ldots, C_n\}$ (where C is an arbitrary clause and $\{C_2, \ldots, C_n\}$ an arbitrary consistent set of clauses) is well-known to be co-NP-complete.

We prove our proposition by a polynomial reduction of *implication between atomic invariant expressions* to this problem.

Let ie be an atomic invariant expression, i.e. an atomic code call condition: it takes as input, a set of clauses, and returns as output, all valuations that satisfy that set of clauses. Let $\mathrm{ANS}(\mathsf{ie}(\{C\}))$ denote the set of results of evaluating ie on C with respect to a state S. Then

$$\mathrm{ANS}(\mathsf{ie}(\{C\})) \subseteq \mathrm{ANS}(\mathsf{ie}(\{C_2, \ldots, C_n\})) \textit{ if and only if } \{C_2, \ldots, C_n\} \models C.$$

Hence, checking whether an arbitrary atomic invariant expression ie_1 implies another atomic invariant expression ie_2 is co-NP hard. □

Lemma A.1 (Translation into predicate logic). There is a translation $\mathfrak{Trans}$ *from simple invariants* INV_{simple} *into predicate logic with equality such that the following holds*

$$\mathcal{I} \models \mathsf{inv} \textit{ if and only if } \mathfrak{Trans}(\mathcal{I}) \cup T_{ord} \models \mathfrak{Trans}(\mathsf{inv}),$$

where T_{ord} *is the theory of strict total orders* $<$ *and* $a \le b$, *(resp.* $a \ge b$*), is an abbreviation for "*$a < b \vee a = b$*", (resp. "*$a > b \vee a = b$*").*

Moreover, a simple invariant "$\mathsf{ic}_1 \Longrightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}'_1$*" is translated into a formula of the form*

$$\underline{\forall}\,(ic_1 \to \forall x(pred_{\langle d_1,f_1\rangle}(\ldots,x) \to (pred_{\langle d_2,f_2\rangle}(\ldots,x)))$$

where $\underline{\forall}$ *denotes the universal closure with respect to all remaining variables. This is a universally quantified formula.*

Proof. We translate each simple invariant to a predicate logic formula by induction on the structure of the invariant.

Code Calls: For each n-ary code call $\mathrm{d}:f(\ldots)$ we introduce a $(n+1)$-ary predicate $pred_{\langle d,f\rangle}(\ldots,\cdot)$. Note that we interpret $\mathrm{d}:f(\ldots)$ as a set of elements. The additional argument is used for containment in this set.

Atomic ccc's: We then replace each simple invariant expression

$$\mathbf{in}(\mathtt{X}, \mathrm{d}_1:f_1(\ldots)) \subseteq \mathbf{in}(\mathtt{Y}, \mathrm{d}_2:f_2(\ldots))$$

by the universal closure (with respect to all base variables) of the formula

$$\forall x(pred_{\langle d_1,f_1\rangle}(\ldots,x) \to pred_{\langle d_2,f_2\rangle}(\ldots,x))$$

Simple Ordinary Invariants: A simple ordinary invariant of the form

$$\mathsf{ic}_1 \Longrightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}'_1$$

is translated into

$$\underline{\forall}\,(\mathsf{ic}_1 \to \forall x(pred_{\langle d_1,f_1\rangle}(\dots,x) \to (pred_{\langle d_2,f_2\rangle}(\dots,x)))$$

where $\underline{\forall}$ denotes the universal closure with respect to all remaining variables.

Simple Invariants: A simple invariant of the form

$$(C_1 \vee C_2 \vee \dots C_m) \Longrightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}'_1$$

is translated into the following m statements $(1 \leq i \leq m)$

$$\underline{\forall}\,(C_i \to \forall x(pred_{\langle d_1,f_1\rangle}(\dots,x) \to (pred_{\langle d_2,f_2\rangle}(\dots,x)))$$

where $\underline{\forall}$ denotes the universal closure with respect to all remaining variables.

Note that according to the definition of a simple ordinary invariant and according to the definition of a code call condition (in front of Example 2.1), ic_1 and the C_i are conjunctions of equalities $s = t$ and inequalities $s \leq t$, $s \geq t$, $s < t$, $s > t$ where s, t are real numbers or variables.

The statement

$$\mathcal{I} \models \mathsf{inv} \textit{ if and only if } \mathfrak{Trans}(\mathcal{I}) \cup T_{ord} \models \mathfrak{Trans}(\mathsf{inv})$$

is easily proved by structural induction on simple invariants and condition lists. □

Proof of Lemma 4.7. We use the translation of Lemma A.1.

The assumption $\not\models \mathsf{inv}_2$ expresses that there is a state S_0 and a substitution θ of the base variables in inv_2 such that $S_0 \models \mathsf{ic}_2\theta$ and there is an object a such that $S_0 \models pred_{\langle d_2,f_2\rangle}(\dots,a)\theta$ and $S_0 \not\models pred_{\langle d'_2,f'_2\rangle}(\dots,a)\theta$.

As inv_1 entails inv_2, inv_1 is not satisfied by S_0. Thus there is θ' such that $S_0 \models \mathsf{ic}_1\theta'$ and there is an object a' with $S_0 \models pred_{\langle d_1,f_1\rangle}(\dots,a')\theta$ and $S_0 \not\models pred_{\langle d'_1,f'_1\rangle}(\dots,a')\theta$.

Now suppose $\langle d'_1, f'_1\rangle \neq \langle d'_2, f'_2\rangle$. Then we simply modify the state S_0 (note a state is just a collection of ground code call conditions) so that $S_0 \models pred_{\langle d'_1,f'_1\rangle}(\dots,a')\theta$. We do this for all θ' that are counterexamples to the truth of inv_1. Because $\langle d'_1, f'_1\rangle \neq \langle d'_2, f'_2\rangle$, this modification does not affect the truth of $S_0 \models pred_{\langle d_2,f_2\rangle}(\dots,a)\theta$ and $S_0 \not\models pred_{\langle d'_2,f'_2\rangle}(\dots,a)\theta$. But this is a contradiction to our assumption that inv_1 entails inv_2. Thus we have proved: $\langle d'_1, f'_1\rangle = \langle d'_2, f'_2\rangle$.

Similarly, we can also modify S_0 by changing the extension of $pred_{\langle d_1,f_1\rangle}(\dots,a')\theta$ and guarantuee that inv_1 holds in S_0. So we also get a contradiction as long as $\langle d_1, f_1\rangle \neq \langle d_2, f_2\rangle$. Therefore we have proved that $\langle d_1, f_1\rangle = \langle d_2, f_2\rangle$.

Our second claim follows trivially from $\langle d'_1, f'_1\rangle = \langle d'_2, f'_2\rangle$, and $\langle d_1, f_1\rangle = \langle d_2, f_2\rangle$. □

Proof of Lemma 4.9. Let $\mathsf{inv}_1 : \mathsf{ic}_1 \Longrightarrow \mathsf{ie}_1\ \Re_1\ \mathsf{ie}'_1$ and $\mathsf{inv}_2 : \mathsf{ic}_2 \Longrightarrow \mathsf{ie}_2\ \Re_2\ \mathsf{ie}'_2$. Then by the computation performed by the **Combine_1** algorithm, the derived invariant has the following form

$$\mathsf{inv} : \textbf{simplify}(\mathsf{ic}_1\ \wedge \mathsf{ic}_2) \Longrightarrow \mathsf{ie}_1\ \Re\ \mathsf{ie}'_2,$$

where $\Re$ is determined by Table 1. If $\textbf{simplify}(\mathsf{ic}_1\ \wedge \mathsf{ic}_2) = \textit{false}$ we are done. In this case, there is no state S satisfying a ground instance of $\mathsf{ic}_1\ \wedge \mathsf{ic}_2$.

We assume that we are given a state S of the agent that satisfies inv_1, inv_2 and $\mathcal{I}$. Let $\mathsf{inv}_1\Theta$ and $\mathsf{inv}_2\Theta$ be any ground instances of inv_1 and inv_2. Then, either $\mathsf{ic}_1(\Theta)$

evaluates to false, or $\mathsf{ie}_1(\Theta)\ \Re_1\ \mathsf{ie}'_1(\Theta)$ is true in S. Similarly, either $\mathsf{ic}_2(\Theta)$ is false or $\mathsf{ie}_2(\Theta)\ \Re_2\ \mathsf{ie}'_2(\Theta)$ is true in S.

If either $\mathsf{ic}_1(\Theta)$ or $\mathsf{ic}_2(\Theta)$ evaluates to false, then $(\mathsf{ic}_1 \wedge \mathsf{ic}_2)(\Theta)$ also evaluates to false, and inv is also satisfied. Let's assume both $\mathsf{ic}_1(\Theta)$ and $\mathsf{ic}_2(\Theta)$ evaluate to true. Then so does $(\mathsf{ic}_1 \wedge \mathsf{ic}_2)(\Theta)$, and both $\mathsf{ie}_1(\Theta)\ \Re_1\ \mathsf{ie}'_1(\Theta)$ and $\mathsf{ie}_2(\Theta)\ \Re_2\ \mathsf{ie}'_2(\Theta)$ are true in S, as S satisfies both inv_1 and inv_2. If $\Re = "="$, then both $\Re_1 = "="$ and $\Re_2 = "="$, $\mathsf{ie}'_1 \to \mathsf{ie}_2$ and $\mathsf{ie}_2 \to \mathsf{ie}'_1$ (in all states satisfying $\mathcal{I}$ and $\mathsf{inv}_1, \mathsf{inv}_2$). Then, we have $\mathsf{ie}_1 = \mathsf{ie}'_1 = \mathsf{ie}_2 = \mathsf{ie}'_2$, hence $\mathsf{ie}_1(\Theta) = \mathsf{ie}_2(\Theta)$ is true in S, and Θ satisfies inv. If $\Re = "\subseteq"$, then $\mathsf{ie}'_1 \to \mathsf{ie}_2$ (in all states satisfying $\mathcal{I}$ and $\mathsf{inv}_1, \mathsf{inv}_2$), and we have $\mathsf{ie}_1\ \Re_1\ \mathsf{ie}'_1$, $\mathsf{ie}'_1 \subseteq \mathsf{ie}_2$, $\mathsf{ie}_2\ \Re_2\ \mathsf{ie}'_2$, and $\mathsf{ie}_1(\Theta) \subseteq \mathsf{ie}_2(\Theta)$. As inv is satisfied by any S that also satisfies both inv_1, inv_2 and $\mathcal{I}$, we have $\{\mathsf{inv}_1, \mathsf{inv}_2\} \cup \mathcal{I} \models \mathsf{inv}$. $\square$

PROOF OF LEMMA 4.11. Let $\mathsf{inv}_1 : \mathsf{ic}_1 \Longrightarrow \mathsf{ie}_1\ \Re_1\ \mathsf{ie}'_1$ and $\mathsf{inv}_2 : \mathsf{ic}_2 \Longrightarrow \mathsf{ie}_2\ \Re_2\ \mathsf{ie}'_2$. Then, either **Combine_3** returns **NIL** or the derived invariant has the following form

$$\mathsf{inv} : \mathbf{simplify}(\mathsf{ic}_1 \vee \mathsf{ic}_2) \Longrightarrow \mathsf{ie}_1\ \Re_1\ \mathsf{ie}'_1.$$

In the latter case, $\mathsf{ie}_1 = \mathsf{ie}_2$, $\Re_1 = \Re_2$ and $\mathsf{ie}'_1 = \mathsf{ie}'_2$ as implied by the **Combine_3** algorithm.

We assume that we are given a state S of the agent that satisfies both inv_1 and inv_2. Let $\mathsf{inv}_1\Theta$ and $\mathsf{inv}_2\Theta$ be any ground instances of inv_1 and inv_2. Then, either $\mathsf{ic}_1(\Theta)$ evaluates to false, or $\mathsf{ie}_1(\Theta)\ \Re_1\ \mathsf{ie}'_1(\Theta)$ is true in S. Similarly, either $\mathsf{ic}_2(\Theta)$ is false or $\mathsf{ie}_1(\Theta)\ \Re_1\ \mathsf{ie}'_1(\Theta)$ is true in S. We have four possible cases.

Case 1: Both $\mathsf{ic}_1(\Theta)$ and $\mathsf{ic}_2(\Theta)$ evaluate to false. Then $(\mathsf{ic}_1 \vee \mathsf{ic}_2)$ also evaluates to false, and inv is also satisfied.

Case 2: $\mathsf{ic}_1(\Theta)$ evaluates to false and $\mathsf{ic}_2(\Theta)$ evaluates to true. Since $S \models \mathsf{inv}_2$, $\mathsf{ie}_1(\Theta)\ \Re_1\ \mathsf{ie}'_1(\Theta)$ is true in S. Then S also satisfies inv.

Case 3: $\mathsf{ic}_1(\Theta)$ evaluates to true and $\mathsf{ic}_2(\Theta)$ evaluates to false. In this case, $\mathsf{ie}_1(\Theta)\ \Re_1\ \mathsf{ie}'_1(\Theta)$ is true in S, since S satisfies inv_1. Hence S also satisfies inv.

Case 4: Both $\mathsf{ic}_1(\Theta)$ and $\mathsf{ic}_2(\Theta)$ evaluate to true. Again, since S satisfies both inv_1 and inv_2, $\mathsf{ie}_1(\Theta)\ \Re_1\ \mathsf{ie}'_1(\Theta)$ is true in S and inv is also satisfied. $\square$

PROOF OF PROPOSITION 4.14. Suppose $X_1 \subseteq X_2$ and $\mathsf{inv} \in C_{\mathcal{I}}(X_1)$. We need to show that $\mathsf{inv} \in C_{\mathcal{I}}(X_2)$. By definition of $C_{\mathcal{I}}$, there are five possible cases:

Case 1: $\mathsf{inv} \in \mathcal{I}$, hence $\mathsf{inv} \in C_{\mathcal{I}}(X_2)$ by definition of $C_{\mathcal{I}}$.

Case 2: $\mathsf{inv} \in X_1$. As $X_1 \subseteq X_2$, $\mathsf{inv} \in X_2$. Hence, $\mathsf{inv} \in C_{\mathcal{I}}(X_2)$ by definition of $C_{\mathcal{I}}$.

Case 3: $\mathsf{inv} = \mathbf{Combine\_1}(\mathsf{inv}_1, \mathsf{inv}_2, \mathcal{I})$ where $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I} \cup X_1$. But then $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I} \cup X_2$ as $X_1 \subseteq X_2$. Hence, $\mathsf{inv} \in C_{\mathcal{I}}(X_2)$.

Case 4: $\mathsf{inv} = \mathbf{Combine\_2}(\mathsf{inv}_1, \mathsf{inv}_2)$ where $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I} \cup X_1$. But then $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I} \cup X_2$ as $X_1 \subseteq X_2$. Hence, $\mathsf{inv} \in C_{\mathcal{I}}(X_2)$.

Case 5: $\mathsf{inv} = \mathbf{Combine\_3}(\mathsf{inv}_1, \mathsf{inv}_2)$ where $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I} \cup X_1$. But then $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I} \cup X_2$ as $X_1 \subseteq X_2$. Hence, $\mathsf{inv} \in C_{\mathcal{I}}(X_2)$. $\square$

PROOF OF LEMMA 4.15. Let $\mathsf{inv} \in C_{\mathcal{I}}(C_{\mathcal{I}} \uparrow^{\omega})$. We need to show $\mathsf{inv} \in C_{\mathcal{I}} \uparrow^{\omega}$. By the definition of $C_{\mathcal{I}}$, there are three possible cases:

Case 1: $\mathsf{inv} \in \mathcal{I}$, then $\mathsf{inv} \in C_{\mathcal{I}} \uparrow^{\omega}$ by the definition of $C_{\mathcal{I}}$.

Case 2: $\mathsf{inv} \in C_{\mathcal{I}} \uparrow^{\omega}$, which is trivial.

Case 3: $\mathsf{inv} = \mathbf{Combine\_1}(\mathsf{inv}_1, \mathsf{inv}_2, \mathcal{I})$ (or $\mathsf{inv} = \mathbf{Combine\_2}(\mathsf{inv}_1, \mathsf{inv}_2)$ or $\mathsf{inv} = \mathbf{Combine\_3}(\mathsf{inv}_1, \mathsf{inv}_2)$) such that $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I} \cup (C_{\mathcal{I}} \uparrow^{\omega})$. There exists a smallest integer k_i (i=1,2) such that $\mathsf{inv}_i \in \mathcal{I} \cup (C_{\mathcal{I}} \uparrow^{k_i})$. Let $k := max(k_1, k_2)$. Then, $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I} \cup (C_{\mathcal{I}} \uparrow^{k})$. By definition of $C_{\mathcal{I}}$ and as $\mathcal{I} \subseteq C_{\mathcal{I}} \uparrow^{k}$, $\mathsf{inv} \in C_{\mathcal{I}} \uparrow^{(k+1)}$. Hence, $\mathsf{inv} \in C_{\mathcal{I}} \uparrow^{\omega}$. □

Proof of Lemma 4.16. Suppose $\mathsf{inv} \in C_{\mathcal{I}} \uparrow^{\omega}$. Then, there exists a smallest integer k, such that $\mathsf{inv} \in C_{\mathcal{I}} \uparrow^{k}$. The proof is by induction on k. Let the inductive hypothesis be defined as $\forall k' : 1 \leq k' \leq k$, if $\mathsf{inv} \in C_{\mathcal{I}} \uparrow^{k}$, then $\mathcal{I} \models \mathsf{inv}$.

Base Step: $k = 1$, $\mathsf{inv} \in C_{\mathcal{I}} \uparrow^{1}$, then there are four possible cases: *(1)* $\mathsf{inv} \in \mathcal{I}$, hence $\mathcal{I} \models \mathsf{inv}$, *(2)* $\mathsf{inv} = \mathbf{Combine\_1}(\mathsf{inv}_1, \mathsf{inv}_2, \mathcal{I})$ where $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I}$. As $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I}$, $\mathcal{I} \models \mathsf{inv}_1, \mathsf{inv}_2$. Then, by Lemma 4.9, $\{\mathsf{inv}_1, \mathsf{inv}_2\} \models \mathsf{inv}$. Therefore, $\mathcal{I} \models \mathsf{inv}$. *(3)* $\mathsf{inv} = \mathbf{Combine\_2}$ (iv_1, inv_2) where $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I}$. Since $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I}$, $\mathcal{I} \models \mathsf{inv}_1, \mathsf{inv}_2$. Then, by Lemma 4.10, $\{\mathsf{inv}_1, \mathsf{inv}_2\} \models \mathsf{inv}$. Therefore, $\mathcal{I} \models \mathsf{inv}$. *(4)* $\mathsf{inv} = \mathbf{Combine\_3}$ (iv_1, inv_2) where $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I}$. As $\mathsf{inv}_1, \mathsf{inv}_2 \in \mathcal{I}$, $\mathcal{I} \models \mathsf{inv}_1, \mathsf{inv}_2$. Then, by Lemma 4.11, $\{\mathsf{inv}_1, \mathsf{inv}_2\} \models \mathsf{inv}$. Therefore, $\mathcal{I} \models \mathsf{inv}$.

Inductive Step: $k > 1$. Let $\mathsf{inv} \in C_{\mathcal{I}} \uparrow^{k}$. Then, there exist $\mathsf{inv}_1, \mathsf{inv}_2 \in C_{\mathcal{I}} \uparrow^{(}$ $k-1)$, such that inv is derived by one of **Combine_1**, **Combine_2** or **Combine_3** operators. That is, either $\mathsf{inv} = \mathbf{Combine\_1}(\mathsf{inv}_1, \mathsf{inv}_2, \mathcal{I})$, or $\mathsf{inv} = \mathbf{Combine\_2}(\mathsf{inv}_1, \mathsf{inv}_2)$, or $\mathsf{inv} = \mathbf{Combine\_3}(\mathsf{inv}_1, \mathsf{inv}_2)$. Because this is the only possibility, as $\mathsf{inv} \notin C_{\mathcal{I}} \uparrow^{j}, j < k$, by definition of k. By the inductive hypothesis $\mathcal{I} \models \mathsf{inv}_1$ and $\mathcal{I} \models \mathsf{inv}_2$. By Lemma 4.9, $\{\mathsf{inv}_1, \mathsf{inv}_2\} \models \mathsf{inv}$. Hence, $\mathcal{I} \models \mathsf{inv}$. □

Lemma A.2. We consider predicate logic with equality and a binary predicate symbol $<$. The language also contains arbitrary constants and parameters from the reals.

We consider a special class of formulae, namely universally quantified formulae of the form $\mathsf{ic} \rightarrow P_i(\underline{t}) \rightarrow P_j(\underline{t'})$, where $\underline{t}, \underline{t'}$ are tuples consisting of variables, constants and parameters, P_i are predicate symbols and ic is an invariant condition involving equality, $<$, variables, constants and parameters. We call this class inv-formulae. Let T be a set of inv-formulae.

*The proof system consisting of **(R0)**: $\frac{\phi(\underline{x})}{\phi(\underline{x})\theta}$, where θ is any substitution for the variables in the tupel $\underline{x}$ and ϕ is an inv-formula, and the two inference rules **(R1)** and **(R2)** below is complete for the class of inv-formulae: For each formula $\mathsf{ic} \rightarrow P_i(\underline{t}) \rightarrow P_j(\underline{t'})$ which follows from T, there is an instance of a derived formula which is identical to it. And each derived formula also follows from T.*

$$\textbf{(R1)} \quad \frac{\begin{array}{lcl} \mathsf{ic}_1 & \rightarrow & P_1(\underline{t_1}) \rightarrow P'_1(\underline{t'_1}) \\ \mathsf{ic}_2 & \rightarrow & P_2(\underline{t_2}) \rightarrow P'_2(\underline{t'_2}) \end{array}}{\boldsymbol{simplify}((\mathsf{ic}_1 \wedge \mathsf{ic}_2)\theta) \rightarrow P_1(\underline{t_1})\theta \rightarrow P'_2(\underline{t'_2})\theta} \quad \begin{array}{l} \textit{where } \theta \textit{ is such that} \\ P'_1(\underline{t'_1})\theta = P_2(\underline{t_2})\theta \end{array}$$

$$\textbf{(R2)} \quad \frac{\begin{array}{lcl} \mathsf{ic}_1 & \rightarrow & P_1(\underline{t_1}) \rightarrow P'_1(\underline{t'_1}) \\ \mathsf{ic}_2 & \rightarrow & P_2(\underline{t_2}) \rightarrow P'_2(\underline{t'_2}) \end{array}}{\boldsymbol{simplify}((\mathsf{ic}_1 \vee \mathsf{ic}_2)\theta\gamma) \rightarrow P_1(\underline{t_1})\theta\gamma \rightarrow P'_1(\underline{t'_1})\theta\gamma} \quad \begin{array}{l} \textit{where } P_1(\underline{t_1})\theta = P_2(\underline{t_2})\theta, \\ P'_1(\underline{t'_1})\gamma = P'_2(\underline{t'_2})\gamma \end{array}$$

The ***simplify*** *routine simplifies invariant conditions (containing the binary symbol $<$) wrt. the theory of real numbers in the signature $<$, $=$, and arbitrary constants and parameters in the reals.*

PROOF. The correctness of the system is obvious, as all rules have this property.

The completeness follows by adapting the classical completeness proof of first-order logic and taking into account the special form of the inv-formulae. Let

$$\varphi : \ \mathsf{ic}^* \to P^*(\underline{t^*}) \to P'^*(\underline{t'^*})$$

be a formula that follows from T . Then the set

$$T \cup \{\underline{\exists}(\mathsf{ic}^* \wedge P^*(\underline{t^*}) \wedge \neg P^{*\prime}(\underline{t^{*\prime}}))\}$$

is unsatisfiable (because $T \models \varphi$). Therefore it suffices to show the following claim:

> *Given a set $T \cup \{\varphi\}$ of inv-formulae, whenever $T \not\vdash \varphi$, then $T \cup \{\neg\varphi\}$ is satisfiable.*

Because then the assumption $T \not\vdash \varphi'$ leads to a contradiction. Therefore, taking into account **(R0)**, we can conclude that at least an inv-formula of the form

$$\varphi' : \ \mathsf{ic} \to P(\underline{t}) \to P'(\underline{t'}),$$

with $\varphi = \varphi'\theta$ must be derivable.

The claim can be shown by establishing that each consistent set $T \cup \{\neg\varphi\}$ containing inv-formulae and their negations, can be extended to a *maximally consistent* set Φ_T which *contains witnesses.*[2] For such sets, the following holds: *(1)* $\phi_{T\cup\{\neg\varphi\}} \vdash \gamma$ implies $\gamma\varphi \in \phi_{T\cup\{\neg\varphi\}}$, *(2)* for all γ: $\gamma \in \phi_{T\cup\{\neg\varphi\}}$ or $\neg\gamma \notin \phi_{T\cup\{\neg\varphi\}}$, and *(3)* $\exists x\gamma \in \phi_{T\cup\{\neg\varphi\}}$ implies that there is a term t with $\gamma[\frac{t}{x}] \in \phi_{T\cup\{\neg\varphi\}}$. These properties induce in a natural way an interpretation which is a model of $\phi_{T\cup\{\neg\varphi\}}$. □

PROOF OF THEOREM 4.17. The proof is by reducing the statement into predicate logic using Lemma A.1. We are then in a situation to apply Lemma A.2. Note that the inference rules of Lemma A.2 act on inv-formulae exactly as **Combine1** and **Combine3** on invariants. Therefore there is a bijection between proofs in the proof system described in Lemma A.2 and derivations of invariants using **Combine1** and **Combine3**. □

PROOF OF COROLLARY 4.18. We are reducing the statement to Theorem 4.17. We transform each invariant with $\Re$ = "=", into two separate invariants with $\Re$ = "$\subseteq$".

If inv is of the form $\mathsf{ic} \Rightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}_2$, we are done, because

(1) the set of transformed invariants is equivalent to the original ones, and

(2) although deriving invariants with $\Re$ = "=" is possible (such invariants are contained in the set Taut and new ones will be generated by **Combine_1**[3] and by **Combine_3**), for all such invariants we have also both their $\subseteq$ counterparts (this can be easily shown by induction).

---

[2] This is analogous to the classical Henkin proof of the completeness of first-order logic. In our case the theory T in question contains only finitely many free variables which simplifies the original proof.

[3] see the first line in Table 1

Let's suppose therefore that inv has the form $\mathsf{ic} \Rightarrow \mathsf{ie}_1 = \mathsf{ie}_2$. We know that both $\mathsf{ic} \Rightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}_2$ and $\mathsf{ic} \Rightarrow \mathsf{ie}_1 \supseteq \mathsf{ie}_2$ are entailed by $\mathcal{I}$ and we apply Theorem 4.17 to these cases. We can assume wlog that none of these two invariants is a tautology (otherwise we are done).

Thus there are inv' (for $\subseteq$) and inv'' (for $\supseteq$). We apply Lemma 4.7 and get that inv' (resp. inv'') has the form $\mathsf{ic}' \Rightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}_2$ (resp. $\mathsf{ic}'' \Rightarrow \mathsf{ie}_1 \supseteq \mathsf{ie}_2$). By symmetry ic' is equivalent (in fact, by using a deterministic strategy it can be made identical) to ic''. Thus by our **Combine_2**, there is also a derived invariant of the form

$$\mathsf{ic}' \Rightarrow \mathsf{ie}_1 = \mathsf{ie}_2,$$

and this derived invariant clearly entails $\mathsf{ic} \Rightarrow \mathsf{ie}_1 = \mathsf{ie}_2$ (because inv' entails $\mathsf{ic} \Rightarrow \mathsf{ie}_1 \subseteq \mathsf{ie}_2$ and inv'' entails $\mathsf{ic} \Rightarrow \mathsf{ie}_1 \supseteq \mathsf{ie}_2$). ☐

PROOF OF COROLLARY 4.19.

(1): The proof is by induction on the iteration of the while loop in the **Compute-Derived-Invariants** algorithm. Let the inductive hypothesis be $\forall i \geq 0$ if inv is inserted into X in iteration i, then $\mathcal{I} \models \mathsf{inv}$.

Base Step: For $i = 0$, $\mathsf{inv} \in \mathcal{I}$, $\mathsf{inv} \rightarrow \mathsf{inv}$, hence $\mathcal{I} \models \mathsf{inv}$.

Inductive Step: Let inv be inserted into X in iteration $i > 0$, and $\mathsf{inv} =$ **Combine_1**$(\mathsf{inv}_1, \mathsf{inv}_2, \mathcal{I})$, where inv_1 and inv_2 are inserted into X at step $(i-1)$ or earlier. Then, by the inductive hypothesis, $\mathcal{I} \models \mathsf{inv}_1$ and $\mathcal{I} \models \mathsf{inv}_2$. By Lemma 4.9, $\{\mathsf{inv}_1, \mathsf{inv}_2\} \models \mathsf{inv}$, hence $\mathcal{I} \models \mathsf{inv}$.

(2): First note that the **Compute-Derived-Invariants** algorithm computes and returns $C_{\mathcal{I}} \uparrow^{\omega}$. The result follows from Theorem 4.17 and Corollary 4.18. ☐

B. AXIOMATIC INFERENCE SYSTEM

Equivalence Rules	**Inference Rules**
$A \cup A = A \cap A = A$	$A \subseteq A$
$A \cup \emptyset = A$ and $A \cap \emptyset = \emptyset$	$(A \cap B) \subseteq A$
$(A \cup B) \cup C = A \cup (B \cup C)$ and $(A \cap B) \cap C = A \cap (B \cap C)$	$A \subseteq (A \cup B)$
$A \cup B = B \cup A$ and $A \cap B = B \cap A$	$((A \cup B) \cap \neg B) \subseteq A$
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	$A \subseteq ((A \cap B) \cup \neg B)$
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$	if $A \subseteq B$ and $B \subseteq C$ then $A \subseteq C$
$\neg(\neg(A)) = A$	if $A \subseteq B$ and $C \subseteq D$ then $(A \cap C) \subseteq (B \cap D)$
$\neg(A \cup B) = \neg A \cap \neg B$ and $\neg(A \cap B) = \neg A \cup \neg B$	if $A \subseteq B$ and $C \subseteq D$ then $(A \cap C) \subseteq (B \cup D)$
$A \cup (A \cap B) = A$ and $A \cap (A \cup B) = A$	

C. INVARIANTS FOR THE SPATIAL AND RELATIONAL DOMAINS

$\mathtt{T} = \mathtt{T}' \wedge \mathtt{L} = \mathtt{L}' \wedge \mathtt{R} = \mathtt{R}' \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{R}, \mathtt{L}, \mathtt{R})) = \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{L}', \mathtt{R}'))$
$\mathtt{T} = \mathtt{T}' \wedge \mathtt{L} = \mathtt{L}' \wedge \mathtt{R} < \mathtt{R}' \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{R}, \mathtt{L}, \mathtt{R})) \subseteq \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{L}', \mathtt{R}'))$
$\mathtt{T} = \mathtt{T}' \wedge \mathtt{R} = \mathtt{R}' \wedge L > L' \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{R}, \mathtt{L}, \mathtt{R})) \subseteq \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{L}', \mathtt{R}'))$
$\mathtt{T} = \mathtt{T}' \wedge R < R' \wedge L > L' \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}, \mathtt{L}, \mathtt{R})) \subseteq \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{L}', \mathtt{R}'))$
$\mathtt{T} = \mathtt{T}' \wedge \mathtt{B} = \mathtt{B}' \wedge \mathtt{U} = \mathtt{U}' \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}, \mathtt{B}, \mathtt{U})) = \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{B}', \mathtt{U}'))$
$\mathtt{T} = \mathtt{T}' \wedge \mathtt{B} = \mathtt{B}' \wedge \mathtt{U} < \mathtt{U}' \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{B}', \mathtt{U}')) \subseteq \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{B}', \mathtt{U}'))$
$\mathtt{T} = \mathtt{T}' \wedge \mathtt{U} = \mathtt{U}' \wedge \mathtt{B} > \mathtt{B}' \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}, \mathtt{B}, \mathtt{U})) \subseteq \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{B}', \mathtt{U}'))$
$\mathtt{T} = \mathtt{T}' \wedge \mathtt{U} < \mathtt{U}' \wedge \mathtt{B} > \mathtt{B}' \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}, \mathtt{B}, \mathtt{U})) \subseteq \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{B}', \mathtt{U}'))$
$\mathtt{T} = \mathtt{T}' \wedge \mathtt{X} = \mathtt{X}' \wedge \mathtt{Y} = \mathtt{Y}' \wedge \mathtt{Rad} = \mathtt{Rad}' \Longrightarrow$ $\mathbf{in}(\mathtt{Z}, \mathtt{spatial}: \mathit{range}(\mathtt{T}, \mathtt{X}, \mathtt{Y}, \mathtt{Rad})) = \mathbf{in}(\mathtt{W}, \mathtt{spatial}: \mathit{range}(\mathtt{T}', \mathtt{X}', \mathtt{Y}', \mathtt{Rad}'))$
$\mathtt{T} = \mathtt{T}' \wedge \mathtt{X} = \mathtt{X}' \wedge \mathtt{Y} = \mathtt{Y}' \wedge \mathtt{Rad} < \mathtt{Rad}' \Longrightarrow$ $\mathbf{in}(\mathtt{Z}, \mathtt{spatial}: \mathit{range}(\mathtt{T}, \mathtt{X}, \mathtt{Y}, \mathtt{Rad})) \subseteq \mathbf{in}(\mathtt{W}, \mathtt{spatial}: \mathit{range}(\mathtt{T}', \mathtt{X}', \mathtt{Y}', \mathtt{Rad}'))$

Table 7. Invariants for the spatial domain (1)

$\mathtt{T} = \mathtt{T}' \wedge \mathtt{R} \leq \mathtt{R}' \wedge \mathtt{L} \leq \mathtt{L}' \wedge \mathtt{L}' \leq \mathtt{R} \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}, \mathtt{L}, \mathtt{R})) \cup \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{L}', \mathtt{R}'))$ $=$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}, \mathtt{L}, \mathtt{R}'))$
$\mathtt{T} = \mathtt{T}' \wedge \mathtt{R} \geq \mathtt{R}' \wedge \mathtt{L} \geq \mathtt{L}' \wedge \mathtt{L} \leq \mathtt{R}' \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{R}, \mathtt{L}, \mathtt{R})) \cup \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}', \mathtt{L}', \mathtt{R}'))$ $=$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{vertical}(\mathtt{T}, \mathtt{L}', \mathtt{R}))$
$\mathtt{T} = \mathtt{T}' \wedge U \leq U' \wedge B \leq B' \wedge B' \leq U \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{horizontal}(\mathtt{T}, \mathtt{B}, \mathtt{U})) \cup \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{horizontal}(\mathtt{T}', \mathtt{B}', \mathtt{U}'))$ $=$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{horizontal}(\mathtt{T}', \mathtt{B}, \mathtt{U}'))$
$\mathtt{T} = \mathtt{T}' \wedge \mathtt{U} \geq \mathtt{U}' \wedge \mathtt{B} \geq \mathtt{B}' \wedge \mathtt{B} \leq \mathtt{U}' \Longrightarrow$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{horizontal}(\mathtt{T}, \mathtt{B}, \mathtt{U})) \cup \mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{horizontal}(\mathtt{T}', \mathtt{B}', \mathtt{U}'))$ $=$ $\mathbf{in}(\mathtt{Y}, \mathtt{spatial}: \mathit{horizontal}(\mathtt{T}', \mathtt{B}', \mathtt{U}))$

Table 8. Invariants for the spatial domain (2)

$\mathtt{Rel} = \mathtt{Rel}' \wedge \mathtt{Attr} = \mathtt{Attr}' \wedge \mathtt{Op} = \mathtt{Op}' \wedge \mathtt{V} = \mathtt{V}' \Longrightarrow$ $\mathbf{in}(\mathtt{X}, \mathrm{rel}: \mathit{select}(\mathtt{Rel}, \mathtt{Attr}, \mathtt{Op}, \mathtt{V})) = \mathbf{in}(\mathtt{Y}, \mathrm{rel}: \mathit{select}(\mathtt{Rel}', \mathtt{Attr}', \mathtt{Op}', \mathtt{V}'))$
$\mathtt{Rel} = \mathtt{Rel}' \wedge \mathtt{Attr} = \mathtt{Attr}' \wedge \mathtt{Op} = \mathtt{Op}' = \text{"}\leq\text{"} \wedge \mathrm{V} < \mathrm{V'} \Longrightarrow$ $\mathbf{in}(\mathtt{X}, \mathrm{rel}: \mathit{select}(\mathtt{Rel}, \mathtt{Attr}, \mathtt{Op}, \mathtt{V})) \subseteq \mathbf{in}(\mathtt{Y}, \mathrm{rel}: \mathit{select}(\mathtt{Rel}', \mathtt{Attr}', \mathtt{Op}', \mathtt{V}'))$
$\mathtt{Rel} = \mathtt{Rel}' \wedge \mathtt{Attr} = \mathtt{Attr}' \wedge \mathtt{Op} = \mathtt{Op}' = \text{"}\geq\text{"} \wedge \mathrm{V} > \mathrm{V'} \Longrightarrow$ $\mathbf{in}(\mathtt{X}, \mathrm{rel}: \mathit{select}(\mathtt{Rel}, \mathtt{Attr}, \mathtt{Op}, \mathtt{V})) \subseteq \mathbf{in}(\mathtt{Y}, \mathrm{rel}: \mathit{select}(\mathtt{Rel}', \mathtt{Attr}', \mathtt{Op}', \mathtt{V}'))$
$\mathtt{Rel} = \mathtt{Rel}' \wedge \mathtt{Attr} = \mathtt{Attr}' \wedge \mathtt{V1} = \mathtt{V1}' \wedge \mathtt{V2} = \mathtt{V2}' \Longrightarrow$ $\mathbf{in}(\mathtt{X}, \mathrm{rel}: \mathit{rngselect}(\mathtt{Rel}, \mathtt{Attr}, \mathtt{V1}, \mathtt{V2})) = \mathbf{in}(\mathtt{Y}, \mathrm{rel}: \mathit{rngselect}(\mathtt{Rel}', \mathtt{Attr}', \mathtt{V1}', \mathtt{V2}'))$
$\mathtt{Rel} = \mathtt{Rel}' \wedge \mathtt{Attr} = \mathtt{Attr}' \wedge \mathrm{V1} \geq \mathtt{V1}' \wedge \mathtt{V2} \leq \mathtt{V2}' \Longrightarrow$ $\mathbf{in}(\mathtt{X}, \mathrm{rel}: \mathit{rngselect}(\mathtt{Rel}, \mathtt{Attr}, \mathtt{V1}, \mathtt{V2})) \subseteq \mathbf{in}(\mathtt{Y}, \mathrm{rel}: \mathit{rngselect}(\mathtt{Rel}', \mathtt{Attr}', \mathtt{V1}', \mathtt{V2}'))$
$\mathtt{Rel} = \mathtt{Rel}' \wedge \mathtt{Attr} = \mathtt{Attr}' \wedge \mathrm{V1} \leq \mathtt{V1}' \wedge \mathtt{V2} \leq \mathtt{V2}' \wedge \mathtt{V1}' \leq \mathrm{V2} \Longrightarrow$ $\mathbf{in}(\mathtt{X}, \mathrm{rel}: \mathit{rngselect}(\mathtt{Rel}, \mathtt{Attr}, \mathtt{V1}, \mathtt{V2})) \cup \mathbf{in}(\mathtt{Y}, \mathrm{rel}: \mathit{rngselect}(\mathtt{Rel}', \mathtt{Attr}', \mathtt{V1}', \mathtt{V2}'))$ $=$ $\mathbf{in}(\mathtt{Z}, \mathrm{rel}: \mathit{rngselect}(\mathtt{Rel}, \mathtt{Attr}, \mathtt{V1}, \mathtt{V2}'))$
$\mathtt{Rel} = \mathtt{Rel}' \wedge \mathtt{Attr} = \mathtt{Attr}' \wedge \mathrm{V1} \geq \mathtt{V1}' \wedge \mathtt{V2} \geq \mathtt{V2}' \wedge \mathrm{V1} \leq \mathtt{V2}' \Longrightarrow$ $\mathbf{in}(\mathtt{X}, \mathrm{rel}: \mathit{rngselect}(\mathtt{Rel}, \mathtt{Attr}, \mathtt{V1}, \mathtt{V2})) \cup \mathbf{in}(\mathtt{Y}, \mathrm{rel}: \mathit{rngselect}(\mathtt{Rel}', \mathtt{Attr}', \mathtt{V1}', \mathtt{V2}'))$ $=$ $\mathbf{in}(\mathtt{Z}, \mathrm{rel}: \mathit{rngselect}(\mathtt{Rel}, \mathtt{Attr}, \mathtt{V1}', \mathtt{V2}))$

Table 9. Invariants for the relational domain (1)

$\texttt{Rel} = \texttt{Rel}' \wedge \texttt{Attr1} = \texttt{Attr1}' \wedge \texttt{Attr2} = \texttt{Attr2}' \wedge \texttt{V1}' = \texttt{V1} \wedge$ $\texttt{V2} = \texttt{V2}' \wedge \texttt{V3}' = \texttt{V3} \wedge \texttt{V4} = \texttt{V4}' \Longrightarrow$ $\mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr1}, \texttt{V1}, \texttt{V2})) \cap \mathbf{in}(\texttt{Y}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}, \texttt{V4}))$ $=$ $\mathbf{in}(\texttt{Z}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}', \texttt{Attr1}', \texttt{V1}', \texttt{V2}')) \cap \mathbf{in}(\texttt{W}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}', \texttt{V4}'))$
$\texttt{Rel} = \texttt{Rel}' \wedge \texttt{Attr1} = \texttt{Attr1}' \wedge \texttt{Attr2} = \texttt{Attr2}' \wedge \texttt{V1}' \leq \texttt{V1}$ $\wedge\ \texttt{V2} \leq \texttt{V2}' \wedge \texttt{V3}' \leq \texttt{V3} \wedge \texttt{V4} \leq \texttt{V4}' \Longrightarrow$ $\mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr1}, \texttt{V1}, \texttt{V2})) \cap \mathbf{in}(\texttt{Y}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}, \texttt{V4}))$ $\subseteq$ $\mathbf{in}(\texttt{Z}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}', \texttt{Attr1}', \texttt{V1}', \texttt{V2}')) \cap \mathbf{in}(\texttt{W}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}', \texttt{V4}'))$
$\texttt{Rel} = \texttt{Rel}' \wedge \texttt{Attr1} = \texttt{Attr1}' \wedge \texttt{Attr2} = \texttt{Attr2}' \wedge \texttt{V1}' \leq \texttt{V1} \wedge$ $\texttt{V2}' \leq \texttt{V2} \wedge \texttt{V3} \leq \texttt{V3}' \wedge \texttt{V4} \leq \texttt{V4}' \wedge \texttt{V1} \leq \texttt{V2}' \wedge \texttt{V3}' \leq \texttt{V4} \Longrightarrow$ $(\mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr1}, \texttt{V1}, \texttt{V2})) \cap \mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}, \texttt{V4}))) \cup$ $(\mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}', \texttt{Attr1}', \texttt{V1}', \texttt{V2}')) \cap \mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}', \texttt{V4}')))$ $\subseteq$ $\mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr1}, \texttt{V1}', \texttt{V2})) \cap \mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}, \texttt{V4}'))$
$\texttt{Rel} = \texttt{Rel}' \wedge \texttt{Attr1} = \texttt{Attr1}' \wedge \texttt{Attr2} = \texttt{Attr2}' \wedge \texttt{V1} \leq \texttt{V1}'$ $\wedge\ \texttt{V2} \leq \texttt{V2}' \wedge \texttt{V3} \leq \texttt{V3}' \wedge \texttt{V4} \leq \texttt{V4}' \wedge \texttt{V1}' \leq \texttt{V2} \wedge \texttt{V3}' \leq \texttt{V4} \Longrightarrow$ $(\mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Attr1}, \texttt{V1}, \texttt{V2})) \cap \mathbf{in}(\texttt{Y}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}, \texttt{V4}))) \cup$ $(\mathbf{in}(\texttt{Z}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}', \texttt{Attr1}', \texttt{V1}', \texttt{V2}')) \cap \mathbf{in}(\texttt{W}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}', \texttt{V4}')))$ $\subseteq$ $\mathbf{in}(\texttt{X}', \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr1}, \texttt{V1}, \texttt{V2}')) \cap \mathbf{in}(\texttt{Y}', \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}, \texttt{V4}'))$
$\texttt{Rel} = \texttt{Rel}' \wedge \texttt{Attr1} = \texttt{Attr1}' \wedge \texttt{Attr2} = \texttt{Attr2}' \wedge \mathrm{V1} \leq \texttt{V1}' \wedge$ $\texttt{V2} \leq \texttt{V2}' \wedge \texttt{V3}' \leq \texttt{V3} \wedge \texttt{V4}' \leq \texttt{V4} \wedge \texttt{V1}' \leq \mathrm{V2} \wedge \texttt{V3} \leq \texttt{V4}' \Longrightarrow$ $(\mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr1}, \texttt{V1}, \texttt{V2})) \cap \mathbf{in}(\texttt{Y}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}, \texttt{V4}))) \cup$ $(\mathbf{in}(\texttt{Z}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}', \texttt{Attr1}', \texttt{V1}', \texttt{V2}')) \cap \mathbf{in}(\texttt{W}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}', \texttt{V4}')))$ $\subseteq$ $\mathbf{in}(\texttt{X}', \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr1}, \texttt{V1}, \texttt{V2}')) \cap \mathbf{in}(\texttt{Y}', \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}', \texttt{V4}))$
$\texttt{Rel} = \texttt{Rel}' \wedge \texttt{Attr1} = \texttt{Attr1}' \wedge \texttt{Attr2} = \texttt{Attr2}' \wedge \texttt{V1}' \leq \texttt{V1} \wedge$ $\texttt{V2}' \leq \texttt{V2} \wedge \texttt{V3}' \leq \texttt{V3} \wedge \texttt{V4}' \leq \texttt{V4} \wedge \texttt{V1} \leq \texttt{V2}' \wedge \texttt{V3} \leq \texttt{V4}' \Longrightarrow$ $(\mathbf{in}(\texttt{X}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr1}, \texttt{V1}, \texttt{V2})) \cap \mathbf{in}(\texttt{Y}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}, \texttt{V4}))) \cup$ $(\mathbf{in}(\texttt{Z}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}', \texttt{Attr1}', \texttt{V1}', \texttt{V2}')) \cap \mathbf{in}(\texttt{W}, \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}', \texttt{V4}')))$ $\subseteq$ $\mathbf{in}(\texttt{X}', \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr1}, \texttt{V1}', \texttt{V2})) \cap \mathbf{in}(\texttt{Y}', \mathrm{rel}: \mathit{rngselect}(\texttt{Rel}, \texttt{Attr2}, \texttt{V3}', \texttt{V4}))$

Table 10. Invariants for the relational domain (2)