TRUTH AND ACTION OSMOSIS (THE *TAO* COMPUTATION MODEL)

ANTÓNIO PORTO, VASCO T. VASCONCELOS
Departamento de Informática, Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

We propose a computation model based on the separation between truth and action. The state of an agent is considered to consist of two components: the task, representing actions to be performed, and the database, representing certain truths. The task specifies a composition (parallel, sequential, choice, synchronous) of elementary actions: queries and commands. Queries may only be executed when certain truths are entailed by the database. Commands always execute and as an effect may change the database. We thus have the osmosis metaphor: the task and database are separated as if by a membrane through which information flows in both directions causing the system to evolve. Truth may enable actions (queries) to be accomplished, and actions (commands) may change what is true. Commands are assumed to have a deterministic nature. We discuss the fundamental semantic properties needed for accommodating this basic model, and consider extensions to accomodate named tasks and their refinement rules, variables in tasks, and systems of named agents with separate rules and databases.

1 Introduction

This paper presents and discusses a new abstract computation model, *TAO*, representing an evolution from ideas present in the *AbstrAct* scheme. [1]

TAO emphasizes the coordination aspects of computing, i.e. the creation, communication and synchronization of processes and agents. The model is based on a state split into a task and a database. The task expresses a certain composition of subtasks, ultimately a composition of elementary actions that test and update the database. These elementary actions are defined logically, through the semantics of the space of situations denoted by the changing database.

The aims behind the development of *TAO* can be summarized as follows.
Abstract view of computation. The least possible commitments are made regarding structural assumptions. For example, no particular data structures are assumed, and arbitrary computations without side-effects are abstracted through a notion of entailment from the database. *TAO* is therefore a very abstract model. The concretization of its abstract assumptions can lead to a concrete (executable) specification language, and further constraints can lead to concrete programming languages.
General applicability. *TAO* is meant to be usable for modeling arbitrary com-

putational systems, and as such addresses issues of concurrency, interaction, change, and locality. Declarative computations can be abstracted away, as mentioned above. *TAO*-derived languages can be used either to specify existing software modules or to program new software pieces, and then glue them together.
Direct support for high-level abstractions. The need is felt for the high-level expression of coordination policies, without having to resort to low-level mechanisms. For example, semaphores or termination flags can be avoided by using general synchronization and sequential operators.
Orthogonal primitive computational ingredients. Though aiming at high-level support, *TAO*'s constructs correspond to simple primitive notions, independent of one another. For example, a general sequential composition operator is available, in contrast to a prefix operator as in the π-calculus that combines notions of input, sequentiality and scoping.
Support for reasoning. The model is built on a firm semantic basis that in principle allows for a suitable program logic in which to carry out program verification. This development is not yet done, however.

The paper starts with an informal presentation of the main ideas behind the design of *TAO*, followed by a discussion of its place in the large space of competing models and languages. The bulk of the paper is a detailed stepwise technical presentation of *TAO*. It starts with the semantic characterization of situations, from which the operational semantics is ultimately built. Then we present the syntax and operational semantics for a simplified version of *TAO*, highlighting the forms of task composition. This is then enhanced by the introduction of variables and terms, allowing for data flow among processes. Finally a structure of agents is built into *TAO*, which allows for rule, task and database encapsulation, task delegation among agents, and dynamic agent creation. The paper ends with some comments on further work that needs to be done.

2 A motivated informal summary of *TAO*

One of our first design principles is to have a *goal directed* view of the behaviour of computational agents. This is in contrast to a purely *reactive* view, and it is our contention that goal directedness not only subsumes reactivity but is intrinsically more modular and efficient. The typical reactive way of describing actions is via $\langle Condition, Action\rangle$ rules. The behaviour of a system obeying such rules can be captured in a goal directed framework by imagining the system as an agent which is executing a task, say react, for which (recursive) rules of the form ($\mathsf{react} \leftarrow Condition \cdot Action \cdot \mathsf{react}$) are given. At first sight this seems to be just a syntactic complication, but instead it has many important consequences. First, dynamic changes of behaviour are easily coded. Just add, for example, a rule such as $(\mathsf{react} \leftarrow \mathsf{cond1} \cdot \mathsf{react1})$, and now $\mathsf{react1}$ has its own rules for controlling system behaviour (which may eventually revert back to react). This is harder to do in the reactive setup, as we have to introduce appropriate flags in the agent's state to control the applicability of *all* the rules, whose conditions and actions have to be modified to accommodate the testing and setting of those flags. We can see the difference in modularity. Second, there is no need to search all rules at all times but only those for the enabled tasks, which can be directly indexed on the tasks' names. This can dramatically affect efficiency. Finally, the ability to name complex tasks, provided by the rule style, opens up the possibility of composing tasks in rules at multiple abstraction levels, corresponding to the structured modelling of behaviours needed in real applications.

We also want to have, nevertheless, the notion of an evolving passive store of information, where conditions for action can be tested, and which can be updated by actions. In other words, we want the possibility of *asynchronous* communication among parallel tasks via a shared dataspace. The motivation is again the question of good matching with the modelling of applications. A quite natural perception of interactions in the world is that they come about through change and reaction to change on a common environment.

Given this, we consider the state of an agent to be composed of two components: an *active* component which we call the task, and a *passive* one which we call the database. The task represents the actions the agent has to perform, i.e. a process that has to unfold, whereas the database represents what is true in a particular state, i.e. a particular situation (a state of partial information). Their intrinsically different nature calls for distinct structural assumptions. We want the task to express temporal constraints on the execution of actions, so we have to consider it as a composition of elementary actions through a suitable set of operators. On the other hand, we want the database to convey a logic notion of truth, used to define the elementary actions, and this can be done without committing to any particular syntax or structure. The space of all situations denoted by the changing database can be viewed abstractly as a constraint system where the fundamental notions of *consistency* and *entailment* are defined.

A crucial design decision is the definition of elementary actions. We consider two types, queries and commands, defined in purely logical terms upon the semantics of situations. Queries check for entailment in the current situation, and commands perform an update of the current situation. They represent the truly basic operational notions: passively waiting for something to happen, and actively making something happen. We consider that updates are necessarily deterministic, so that all non-determinism in a system may be explicitly controlled and not hidden inside elementary actions. This also simplifies the implementation.

Tasks are built by composing elementary actions. We consider it essential to have general composition operators for parallelism, sequentiality, synchronization and choice, whose combinations allow the high-level expression of many interesting control regimes.

The task composition operators by themselves are not enough for having abstraction power. It must also be possible to name composed tasks, and to use those named tasks in further compositions. This is achieved by having action rules, which express the decomposition of named tasks. Rules are also the vehicle for expressing recursion and thereby tasks corresponding to infinite processes.

Any practical computational system must do some data processing and data transfer. Data processing actions can be abstracted away as queries, but we need to represent the data itself and place-holders for its transfer. Given the abstract nature of *TAO* we assume a universe of terms without any structural assumptions except the following. There is a subset of terms which we call variables, whose occurrences inside terms we can classify as free or bound, and the result is defined of applying substitutions of variables by terms in terms. The syntax of tasks includes variables, which are classified in context as input or flexible, this leading to an operational model where queries are the source of substitutions for variables, that act purely as data flow channels, and not as a means for process synchronization. This stems from our principle of orthogonal ingredients.

For dealing with notions of locality at all levels we introduce the concept of agent, which has a unique name and its own local set of rules, task and database. There is a special agent creation task which handles the unique name creation and scoping, and a delegated task construct which behaves as a remote procedure call between agents. This gives us a more direct handle for modelling locality, point-to-point communication and delegation.

3 *TAO* in perspective

TAO is a computational model whose emphasis is on the *coordination* of processes and agents. It belongs to the broad class of coordination models based on generative communication through a shared dataspace, such as Linda [2] (with variant embeddings like Shared Prolog [3]), GAMMA [4] and *LO*. [5] The use of generic ask and tell operations as primitive actions establishes a connection to concurrent constraint programming (CCP).[6] The goal-driven operation and the use of shared logic variables further link it to the family of concurrent logic languages (CLL).[7] Finally, it shares with process calculi such as CCS[8] and the π-calculus[9] the explicit use of various composition operators for processes.

Of course, several features of *TAO* make it different from its relatives. We try in this section to summarize those differences, thus putting *TAO* in perspective among other concurrent and coordination languages and models.

Coordination and computation

"Coordination" and "computation" are sometimes used as opposing terms, relating respectively to issues of communication and control among processes and to issues of data processing. Coordination models such as Linda, GAMMA and *LO* address exclusively issues of coordination. They assume that data processing is to be carried out by processes programmed in other languages, which coordinate themselves by executing calls to coordination modules.

TAO, in contrast, promotes the view that agent and process coordination is the enveloping layer from within which data processing is called. The model provides for this in two ways: the query action, which is elementary from the coordination point of view, may in fact be an arbitrarily complex data processing activity that builds results from data in the database without changing it (say, compute the shortest path in a graph); named tasks can be linked to arbitrary procedures written in whatever language, parameters being used for data communication. Note that accessing the database from externally defined processes, as in the other models, remains a possibility.

The level of control

In our view coordination is about control. A crucial question is at what level can this be exercised, which depends on the available coordination mechanisms. Ideally we want coordination languages to allow us to express control in ways as close as possible to our intuitive high-level modelling of the systems we want to build.

Using Linda all control is local and low-level, as only the dataspace access primitives are made available and the emerging behaviour of systems using these is nowhere globally expressed. GAMMA takes the level of control a bit higher, through the expression of reactive rules, but this remains a minimalist approach regarding high-level control. Recent work on schedules for GAMMA[10] has been spurred by the realization that in practice more control is needed. *LO* uses also reactive rules but with a more structured dataspace, allowing the expression of notions such as dynamic multiple object spaces with broadcast facilities.

High-level control in *TAO* is achieved through a sufficiently rich set of process composition operators and the abstraction facilities provided by named tasks and corresponding rules.

A case in point is the expression of sequentiality. In trying to keep a close gap between our high-level modelling of systems and the programs to implement them, a general sequentiality composition operator seems a must and was included in *TAO*, though it is conspicuously absent from most other related models. In these, sequentiality can be achieved indirectly through the manipulation of the dataspace, but this may make it hard to see the sequentiality in non-trivial cases, thereby impairing our ability to reason about programs. The same is true of CLL, CCP and CCS, where only a restricted form of sequentiality is available in the form of guards, ask and prefix, respectively.

Another control feature of *TAO* which is even more unique is the synchronous composition operator. Its existence is again in the spirit of providing high-level control facilities, as it allows the easy expression of solutions to mutual exclusion problems without having to program semaphores or monitors. The burden is pushed away from the programmer and into the implementation.

The split state

A landmark of *TAO* is the splitting of the state into two distinct components: the task, expressing the processes to be executed, and the database, expressing facts which can be queried and updated by processes. This is exactly the stance also of CCP, with the distinction between the process expression and the constraint store.

In CLL the state is just the process expression, within whose structure all data is bound. The same situation is true of CCS-like process calculi.

In Linda and GAMMA the state is a uniform multiset of data tokens, and processes are expressed by independent external entities acting on those tokens. The proposal for higher-order GAMMA [11] caters for the uniform representation of processes as well as data on the dataspace. *LO* has a dataspace structured as multiple multisets, but within each one data is uniform; processes reflect external rules which make no distinctions among the dataspace tokens.

The logic basis of action

In Linda, GAMMA and *LO* we find the primitive actions of retracting and adding data tokens to the dataspace. Such a dataspace can be assigned a logical meaning through linear logic,[12] which is indeed the theoretical basis behind *LO*.

One of the most distinguishing aspects of *TAO* is its very different approach to the dataspace and the elementary actions. We avoid committing to a particular structure such as a multiset. Instead, the database is taken to denote an abstract situation with no structural assumptions, the space of situations having an intrinsic abstract logic in terms of entailment and consistency. We assume just a partial order of entailment among situations such that the situation space is consistently complete. The primitive actions are the queries, corresponding to checking entailment, and the commands, corresponding to imposing entailment. In order for the latter to be deterministic in all cases the situation space is further assumed to be coherent.

This logical view of the dataspace and the primitive actions is very much in accordance to the one of CCP. The crucial difference is that in CCP the tell operation includes a consistency test, so that the store changes only monotonically, whereas a *TAO* command always executes, if necessary by removing inconsistencies, thereby causing non-monotonic change. Recently an extension of CCP with non-monotonic change has been proposed,[13] but based on assertion/deletion of syntactic tokens rather than on the abstract logic properties of the store, as we do.

Queries and commands make a big difference with respect to retracting and adding data tokens. Each *TAO* database is seen as having logic consequences, which can be exploited by tests of arbitrary complexity. This gives us the ability to wrap up complex non-destructive data processing operations which can be implemented by external modules. Commands are the source of change but only as a contextual effect of being applied in a situation inconsistent with what the command imposes. For example, commanding a zero-one toggle to be zero either changes it from one or leaves it at zero; we cannot perform in *LO* the analogous multiset operation of deleting a token if present, otherwise do nothing. Commands, though limited in expressivity by the coherence requirement, can still express global conditions. We can for example turn a whole database relation to empty with one command, whereas the analogous multiset operation of clearing all tokens of a given type is not expressible as an atomic operation in *LO*. These comments again stress the ability of *TAO* to express high-level control.

Logic variables

Besides the communication of processes through the database, with its intrinsic space and time decoupling, *TAO* allows also for a tighter form of communication through the use of shared logic variables, which is spatially coupled because of the explicit occurrences of shared variables and is partially time-coupled by the input/flexible restrictions on variable occurrences.

This is of course reminiscent of the use of shared variables as communication channels in CLL. In these languages, however, this is the only communication medium, so no pure space/time decoupled communication exists. In CLL shared variables are used not only for communication but also for process synchronization. The tighter discipline of input/flexible variables in *TAO* restricts shared variables to a pure communication role. This considerably simplifies the implementation of shared variables, as no concurrent write accesses occur. The stream-based communication promoted by CLL can be easily implemented in *TAO* through stream agents, just one out of many possible communication modes, contrary to CLL.

Multiple agents

For many languages a basic operational model has to be extended in order to cater for a modular layer of agents and their interactions. This could be expected to hold for *TAO*, specially as the proposed interaction, through remote procedure calls among named agents, seems to be at odds with the basic global communication and synchronization through a shared database. The formal definition shows, however, that we are essentially specializing the basic model. Agents are achieved through structural assumptions on rules, databases and named tasks, and a whole system of agents is still captured by a state with a (split) task, a (split) database and a (split) set of rules.

There is however a real extension of the basic model with the introduction of the agent scoping task, with its essential dynamic properties: automatic creation of new constant (rather than variable) names, and dynamic change of the (split) rule set. These two aspects could have been introduced as orthogonal extensions not necessarily related to multiple agents. The creation of new names is essential and available in many models such as the π-calculus (explicitly) or CLL (implicitly). The dynamics of the rule set is used for changing the set of agents, but it could also be used inside an agent for contextual changes of functionality.

4 The structure of situations

At the heart of the *TAO* model lies the notion of *situation*, that we take to be the semantic denotation of the passive part of the state which we call the database. Situations are the basis for defining the semantics of elementary actions, so we need to discuss first the algebraic structure of the space of situations.

We consider an abstract set of situations partially ordered by an *entailment* relation $\sqsubseteq$. We say that a situation s entails a situation s' whenever $s' \sqsubseteq s$. Remember that situations are semantic entities, so we are considering here a semantic notion of entailment whereby distinct situations cannot entail each other; this explains why we are considering a partial order rather than just a preorder. Intuitively one can interpret situations as states of partial information, and entailment as information subsumption.

Working with partially ordered sets we can make use of the following standard notions of consistency.

Definition 4.1 (Consistent, consistently complete poset) *A subset of a partially ordered set is* consistent *if every one of its finite subsets has an upper bound. A partially ordered set is* consistently complete *if every consistent subset has a least upper bound.*

We consider that situation structures must be non-empty consistently complete partially ordered sets. We call these *cc-posets*.

An important property of cc-posets is the following.

Proposition 4.2 *A cc-poset is a complete lower semilattice.* [a]

Proof. Take any non-empty subset S of a cc-poset, and consider the set L of all lower bounds of S. L is clearly consistent, given that S is non-empty, so there is a least upper bound u of L. As any element in S is also an upper bound of L, u must be a lower bound of S and therefore belong to L. Being an upper bound of L, it must be in fact its greatest element, i.e. S has a greatest lower bound u. ∎

This translates into the following properties of cc-posets. The least element always exists, and corresponds to no information. Two elements s and s' always have a greatest lower bound $s \sqcap s'$, corresponding to the information common to both, and have a least upper bound $s \sqcup s'$ if they are consistent, corresponding to the joint compatible information contained in either of them.

This is however still too general for characterizing situation structures, due to the problem of defining updates. Imagine that we have a current situation s expressing some information, and are given another situation s' expressing some information that we want to enforce upon s. The update problem is to define the new updated situation u. We clearly want u to entail s', but we have two further requirements upon u: it must result from a minimal change of s, and it must be uniquely determined. For many cc-posets the uniqueness of updates cannot be defined. Just think of a cc-poset with all the

[a] Every non-empty subset has a greatest lower bound.

consistent equality/inequality constraints over three items a, b and c. Given $s = \{a = b = c\}$ and $s' = \{a \neq b\}$ there are two equally likely candidates for u, namely $\{a \neq b, b \neq c, a = c\}$ and $\{a \neq b, a \neq c, b = c\}$.

The following definition captures the idea that all updates satisfy our two requirements of uniqueness and minimal change.

Definition 4.3 (Definite updates) *A cc-poset has* definite updates *if for every pair of elements s, s' there is a greatest lower bound l of s consistent with s'. We say the least upper bound u of l and s' is the* update *of s by s', and define a corresponding update function $s \triangleright s' = u$.*

This is all we need for defining situation structures.

Definition 4.4 (Situation structure) *A* situation structure *is a cc-poset with definite updates.*

Now this is a definition with too much of an operational flavour, which does not help in understanding what kind of restriction is being put on cc-posets to get situation structures. It turns out that we can relate this definition to a stronger notion of consistency.

Definition 4.5 (Coherence) *A set is pairwise consistent if every subset with two elements is consistent. A cc-poset is* coherent *if every pairwise consistent subset is consistent.*

Theorem 4.6 *For cc-posets, coherence is equivalent to definite updates.*

Proof. We first prove that coherence implies definite updates.

Let S be a coherent cc-poset. Consider an arbitrary pair s, s' of elements of S, and let C be the set of all lower bounds of s consistent with s'. By construction C has an upper bound s, so it is consistent and must have a least upper bound $u \sqsubseteq s$. Now consider the set $C' = C \cup \{s'\}$. By construction s' is consistent with every element of C, and C being consistent is pairwise consistent, so C' is pairwise consistent. Coherence implies that C' is consistent and has a least upper bound u'. As u is the least upper bound of C, u' must be the least upper bound of u and s', otherwise it would not be the least upper bound of C'. This means that u is consistent with s' and must belong to C, being in fact its greatest element. We then conclude that u' is the update of s by s'.

Now we prove by refutation the reverse implication, i.e. we prove that not being coherent implies not having definite updates.

Let S be a cc-poset which is not coherent. Then it must have a subset which is pairwise consistent but not consistent, which in turn means there is an inconsistent (but pairwise consistent) finite subset N of S. Set N must be non-empty, otherwise its inconsistency would mean that S would be empty, and we are assuming cc-posets are non-empty. Then N must have a subset with one element, which is obviously consistent. As N is finite, we can easily conclude that it must have a maximal consistent subset, say M, which therefore has a least upper bound s. Since N is inconsistent, there must be one element s' in N that is not in M. We can now prove by refutation that there is no definite update of s by s'. Suppose there is. Then there must be a greatest lower bound l of s consistent with s'. Now recall that N is pairwise consistent, so s' is consistent with every element of M. These being lower bounds of s consistent with s', l must be an upper bound of M. Given that l has an upper bound u with s', then u is an upper bound of $M \cup \{s'\}$, which is therefore consistent, and this contradicts the hypothesis that M is a maximal consistent subset of N. ■

Coherence gives us some intuitive feel for the sort of restriction being imposed on situation structures. It is a strong restriction, requiring consistency to have a form of locality which lowers the complexity of consistency checking. Indeed, checking whether adding a situation to a consistent set of situations results in a consistent set is linear on the size of the set.

Let us now look at some particular examples of situation structures.

Example 4.7 *The flat situation space.*

Probably the simplest example of a situation structure is to take an arbitrary unordered set and just add a bottom element:

$$\mathcal{S} = X \cup \{\bot\}\ , \qquad \sqsubseteq\ = (\{\bot\} \times X) \cup \{\ \langle s, s\rangle \mid s \in \mathcal{S}\ \}\ .$$

$\mathcal{S}$ is clearly coherent, as consistent subsets have at most two elements. The update function is

$$x \triangleright y = \begin{cases} y & \text{if} \quad y \in X \\ x & \text{if} \quad y = \bot \end{cases}$$

i.e. when restricted to X "update with" is equivalent to "change to". This is the sort of situation structure used in finite state automata, for example. □

Example 4.8 *Partial Herbrand interpretations.*

Take a first-order logic language whose set of ground atoms (the Herbrand base) is $\mathcal{H}$. The partial Herbrand interpretations are the possible sets of consistent pairs of true and false atoms:

$$\mathcal{S} = \{\ \langle T, F\rangle \mid T \cup F \subseteq \mathcal{H},\ T \cap F = \emptyset\ \}.$$

A natural ordering is that based on the amount of information,

$$\langle T, F\rangle \sqsubseteq \langle T', F'\rangle \quad \text{iff} \quad T \subseteq T', F \subseteq F'\ \ ,$$

under which $\mathcal{S}$ is a situation structure.

We can prove this by refutation. If $\mathcal{S}$ is not a situation structure it has a pairwise consistent but inconsistent subset X. So X has a finite subset $\{\langle T_1, F_1\rangle, \ldots, \langle T_n, F_n\rangle\}$ with no upper bound. In particular, $\langle T_1 \cup \cdots \cup T_n, F_1 \cup \cdots \cup F_n\rangle \notin \mathcal{S}$, and this means there is an atom $a \in \mathcal{H}$ belonging to both $T_1 \cup \cdots \cup T_n$ and $F_1 \cup \cdots \cup F_n$, i.e. for some i and j between 1 and n we have $a \in T_i$ and $a \in F_j$. X being pairwise consistent, the pair $s_i = \langle T_i, F_i\rangle$ and $s_j = \langle T_j, F_j\rangle$ has an upper bound $\langle T, F\rangle$ satisfying $T_i \cup T_j \subseteq T$ and $F_i \cup F_j \subseteq F$. But then $a \in T \cap F$, which cannot be.

The update function is

$$\langle T, F\rangle \triangleright \langle T', F'\rangle = \langle (T \setminus F') \cup T', (F \setminus T') \cup F'\rangle \ ,$$

reflecting the "minimal change" operation of throwing away from a situation only that which contradicts the new information being incorporated. $\square$

5 Syntax

Let us shift to syntax. Rather than presenting the full language right away, we concentrate first on the basic aspects of task composition, elementary tasks, named tasks, rules, and databases.

5.1 Tasks

We want the task of an agent to represent a certain composition of actions that have to be performed. What forms of composition are needed? We aim at a general but high-level model for coordinated computing, so the following considerations are in order.

Concurrency We certainly want parallel composition, for expressing the concurrent execution of elementary tasks.

Sequentiality We find that sequential composition is also essential in a practical programming language. Although theoretically we could do away without it, we would then impose quite a burden on the programmer and on the verification process, by having to resort to low-level manipulation of boolean flags in the situation to impose sequentiality constraints.

Non-determinism Another must is to have a choice operation, a form of composition for expressing non-determinism among alternative tasks.

Synchronicity The importance of atomic (i.e. synchronous) combinations of queries and updates has always been recognized for solving mutual exclusion problems in concurrency. The ability to use a general synchronization operator, rather than a fixed set of atomic actions, raises the level of expressiveness of programs and the ease of reasoning about them.

The first three forms of composition are "standard" in the various process calculi. Restricted forms of sequentiality and synchronicity are sometimes bundled in the concept of guard in some languages, such as those in the concurrent logic family.

So, assuming we have a set $\mathcal{C}$ of commands, a set $\mathcal{Q}$ of queries, and a set $\mathcal{A}$ of named tasks, we define the tasks as follows.

Definition 5.1 (Tasks) *The* tasks *are the least set $\mathcal{T}$ satisfying*

null	$\mathbf{0} \in \mathcal{T}$	*synchronous*	$t_1 \diamond t_2 \in \mathcal{T}$
query	$\mathcal{Q} \subset \mathcal{T}$	*sequence*	$t_1 \cdot t_2 \in \mathcal{T}$
command	$\mathcal{C} \subset \mathcal{T}$	*parallel*	$t_1 \mid t_2 \in \mathcal{T}$
named task	$\mathcal{A} \subset \mathcal{T}$	*choice*	$t_1 + t_2 \in \mathcal{T}$

if $t_1, t_2 \in \mathcal{T}$. We shall use the letter t to range over tasks.

The atomic [b] tasks must include the elementary tasks (queries and commands), but we also allow for named tasks. These provide the ability to name and define abstractions of tasks (i.e. procedures), by means of rules, thereby allowing the expression of infinite processes. We deal with elementary and named tasks in turn.

5.2 Elementary tasks

Elementary tasks are queries and commands. We can easily introduce two monadic operators for distinguishing between the two, but these must apply to syntactic entities that refer to situations.

The example of partial Herbrand interpretations as a situation structure suggests the use of a logic language for making statements about situations. The problem is that in general a formula (or a set of formulas) may not have a single partial model but several, thus possibly describing more than one situation. One can of course restrict the language so as to have a single model semantics. One possibility is to consider only *definite* statements that have a least partial model. Another is to consider logic programs under the well-founded semantics, a non-monotonic semantics that makes negative

[b] in the syntactic sense

assumptions. The latter is a possibility for the database of an agent, where negative assumptions may make sense for defining the corresponding situation, but when using queries and commands we want to express truly partial information about a situation, without any extra assumptions. We take the definite statements to be an essential restriction for commands, though not so for queries. The reason has to do with the crucial difference in their respective operational semantics. Queries are passive and are only checked against a *given* current situation, so their possible intrinsic nondeterminism, caused by a non-definite semantics, may be useful for expressing sets of possible situations that we want to satisfy the query. Commands, on the other hand, are active and enforce *one* new situation, so nondeterminism here would really mean an unknown outcome of the operation. As this operation is the basis of change in the system, we do want it to be deterministic. Having a choice composition in tasks, we can always have *explicit* nondeterminism if we so wish, even among commands.

Let us formalize the required notions.

Assumption 5.2 *There is a situation structure $\mathcal{S}$ with least element $\perp$.*

Definition 5.3 (Situation types) *The* situation types $\mathcal{S}^T \subset \wp(\mathcal{S})$ *are the upward closed sets of situations, i.e. for any $S \in \mathcal{S}^T$, if $s \in S$ and $s \sqsubseteq s'$ then $s' \in S$.*

Definition 5.4 (Propositions, consistent propositions) *A* proposition p *is a syntactic entity equipped with a semantics $[\![p]\!] \in \mathcal{S}^T$ yielding a situation type. A proposition is said to be* consistent *if its semantics is non-empty.*

The requirement of upward closure guarantees a truly partial semantics for propositions, in the sense that any consistent extension of a situation satisfying a proposition also satisfies it.

Assumption 5.5 (Query propositions) *There is a set $\mathcal{P}_?$ of* query propositions.

The propositions used for commands have to be more restricted, in order to guarantee definite updates. They should in any case be a particular case of query propositions, as it is reasonable to assume that any condition expressed in a command can also be posed as a query.

Definition 5.6 (Definite situation types) *A situation type S is* definite *if it is empty or has a least element. The least element of a non-empty definite situation type S is denoted by $\widehat{S}$.*

Notice that a single situation s fully characterizes a non-empty definite situation type S, namely the least upward closed set containing s (which is then $\widehat{S}$). This justifies the word 'definite'.

Definition 5.7 (Definite propositions) *A proposition is called* definite *if its semantics yields a definite situation type.*

Assumption 5.8 (Command propositions) *The query propositions have a subset $\mathcal{P}_! \subset \mathcal{P}_?$ of* command propositions *that are definite. There is a distinguished* null *command proposition $\top$ such that $[\![\top]\!] = \mathcal{S}$.*

Notice that $\widehat{[\![\top]\!]} = \perp$, i.e. $\top$ has no information content and can logically be regarded as true. Having defined the query and the command propositions, we may introduce the two monadic operators to distinguish the corresponding tasks.

Definition 5.9 (Queries, commands)

$$\begin{aligned} \textit{queries} \quad & \mathcal{Q} = \{\, ?p \mid p \in \mathcal{P}_? \,\} \\ \textit{commands} \quad & \mathcal{C} = \{\, !p \mid p \in \mathcal{P}_! \,\} \end{aligned}$$

5.3 Named tasks and rules

Named tasks provide for the naming and definition of task abstractions by means of rules.

Assumption 5.10 (Named tasks) *The syntactic vocabulary includes a set $\mathcal{A}$ of* named tasks*. We shall use the letter a to range over named tasks.*

Named tasks have no a-priori meaning. In order to specify an agent, one must provide task refinement rules for the vocabulary of named tasks, i.e. provide procedure definitions.

Definition 5.11 (Task rules) *The* task rules *$\mathcal{R} = \mathcal{A} \times \mathcal{T}$ associate named tasks with tasks. We use the notation $a \leftarrow t$ for a rule $\langle a, t \rangle \in \mathcal{R}$, and call a the rule* head *and t the rule* body*. The letter r will be used to range over sets of rules.*

Given a set of rules, each is seen as specifying a possible way of executing a named task, several rules for the same named task representing alternative paths for its execution. This will be formalized in the next section dealing with the operational semantics.

Notice that named tasks can appear in the bodies of rules, thereby providing the power to express infinite processes.

5.4 Databases

We have argued before that a database must denote a situation, and that this can be achieved for example via logic programs with the well-founded semantics. However, this non-monotonic semantics is not compositional with respect to program union, thereby causing problems for the syntactic realization of updates. This suggests that assumptions will have to be made for all updates to be syntactically realizable; we'll do so in the next section.

Assumption 5.12 (Databases) *There is a set $\mathcal{D}$ of syntactic entities called* databases *whose semantics yield situations, i.e. for any $d \in \mathcal{D}$, $[\![d]\!] \in \mathcal{S}$. There is a distinguished* null *database ∇ such that $[\![\nabla]\!] = \bot$.*

6 Operational semantics

We first look at the elementary operations—queries and commands on the database—, then we define the possible steps that a task may take, and from these the state transitions.

6.1 Deduction and update

Queries are to be executed against a database. Having entailment defined at the semantic level, we now bring it to the syntactic level and assume a corresponding operational counterpart.

Definition 6.1 (Entailment) *A database d* entails *a query proposition p, written $d \models p$, if $[\![d]\!] \in [\![p]\!]$.*

Assumption 6.2 (Deduction) *There is a* deduction *relation $\vdash \subset \mathcal{D} \times \mathcal{P}_?$ between databases and query propositions that is sound and complete with respect to entailment, i.e. $d \vdash p \Leftrightarrow d \models p$.*

For commands we must have a syntactic counterpart to the semantic update. We require syntactic coverage of the possible updates, and uniqueness of the result.

Assumption 6.3 (Database update) *For any database d and any consistent command proposition p there is a database d' such that $[\![d']\!] = [\![d]\!] \triangleright \widehat{[\![p]\!]}$. There is a syntactic* update *function that chooses one such d' for any given database d and consistent command proposition p. We overload the notation and write $d \triangleright p$ for the result of the syntactic update of d by p.*

6.2 Synchronous composition

The availability of synchronous composition in tasks means that we may have to combine several queries and commands in order to perform a single step. Queries and commands are operationally separate concerns, so we have to consider only combinations of queries among themselves, and likewise for commands. The logical interpretation of these combinations is as conjunctions of the respective propositions.

Assumption 6.4 (Conjunction) *The query propositions are closed under an operation of* conjunction $\wedge$*, i.e. for any query propositions p and p' there is a query proposition $p \wedge p'$ such that $[\![p \wedge p']\!] = [\![p]\!] \cap [\![p']\!]$. The subset of command propositions is itself closed for conjunction.*

We can easily prove that this assumption is sound.

Proposition 6.5 *Situation types are closed under intersection, and so are definite situation types.*

Proof. It is easy to verify that the intersection of situation types preserves upward closure and is therefore also a situation type. Now take any two definite situation types x_1 and x_2. If one is empty so is their intersection, which is then a definite situation type. If both are nonempty they have least elements, say s_1 and s_2. Situation types being upward closed, $x_1 \cap x_2$ contains exactly the common upper bounds of both s_1 and s_2. If $x_1 \cap x_2 = \emptyset$ it is definite. If it is non-empty, s_1 and s_2 have an upper bound and are thereby consistent. As the situation structure is consistently complete they have a least upper bound, which must belong to $x_1 \cap x_2$, and is therefore its least element, making it definite. ∎

6.3 Task reduction

A crucial point about the operational semantics is the way in which non-determinism is taken into account. In general, and given any particular state, there may be several paths for an agent to follow, due to choices in tasks. As some tasks may be blocked for a given database, due to failing queries, some control should be exercised before committing to a particular choice. Between the extremes of blind choice and unbounded look-ahead for viable alternatives, we take the (usual) approach of looking ahead for one possible next step that is enabled, and commit to the respective path by executing it.

Definition 6.6 (Step) *A* step *is a pair of a query proposition and a command proposition. Given a database d, a step $\delta = \langle q, c\rangle$ is said to be* enabled *in d whenever the query is deduced ($d \vdash q$) and the command is consistent. In this case δ is said to* update *d to the database $d \triangleright c$.*

For the one-step look-ahead semantics, it is essential to specify what are the possible first steps that a task may reduce through.

Definition 6.7 (Task reduction) *The* reduction $\xrightarrow{\delta}$ *induced on tasks by a step δ is the least relation over $\mathcal{T}^2$ satisfying the following rules.*

$$\frac{}{?p \xrightarrow{\langle p,\top\rangle} \mathbf{0}} \qquad\qquad \frac{}{!p \xrightarrow{\langle \top,p\rangle} \mathbf{0}} \qquad\qquad \text{(Elementary)}$$

$$\frac{t_1 \xrightarrow{\delta} t_1'}{t_1 \mid t_2 \xrightarrow{\delta} t_1' \mid t_2} \qquad \text{(Parallel)}$$

$$\frac{t_1 \xrightarrow{\delta} t_1'}{t_1 \cdot t_2 \xrightarrow{\delta} t_1' \cdot t_2} \qquad \text{(Sequence)}$$

$$\frac{t_1 \xrightarrow{\delta} t_1'}{t_1 + t_2 \xrightarrow{\delta} t_1'} \qquad \text{(Choice)}$$

$$\frac{t_1 \xrightarrow{\langle q_1, c_1 \rangle} \mathbf{0} \qquad t_2 \xrightarrow{\langle q_2, c_2 \rangle} \mathbf{0}}{t_1 \diamond t_2 \xrightarrow{\langle q_1 \wedge q_2, c_1 \wedge c_2 \rangle} \mathbf{0}} \qquad \text{(Synchronous)}$$

You may have noticed that the above definition is asymmetrical for some binary connectives, and does not take care of special cases of reduction to the null task. The rules capture the essence of the reduction. Symmetries and riddance of null tasks are handled by a structural congruence on tasks we define next.

Notice that the synchronization of two tasks is only reducible if both components are reducible to the null task, i.e. are executable in one single step.

6.4 Structural congruence

The congruence relation on tasks captures task equivalence at the static level, simplifying the treatment of state transition.

Definition 6.8 (Structural congruence) *The* structural congruence $\equiv$ *on tasks is the least congruence over* $\mathcal{T}^2$ *satisfying the following rules.*

$$(\mathcal{T}/\equiv, \cdot, \mathbf{0}) \textit{ is a monoid}^{\,c}$$
$$(\mathcal{T}/\equiv, \mid, \mathbf{0}) \textit{ is a commutative monoid}$$
$$(\mathcal{T}/\equiv, +, \mathbf{0}) \textit{ is a commutative monoid}$$

We have said that each rule specifies a possible way of executing a named task, and that several rules for the same named task represent alternative paths for its execution. This can be formalized as follows.

Definition 6.9 (Structural congruence) *The* structural congruence $\equiv_r$ *on tasks induced by a set of rules* r *is the least extension of the structural congruence* $\equiv$ *in definition 6.8 that satisfies*

$$a \equiv_r t + a$$

for each rule $a \leftarrow t \in r$.

[c] That is, the operator $\cdot$ is associative and has a neutral element $\mathbf{0}$.

This congruence allows the unfolding of a named task into a choice containing all the bodies of rules associated with the named action. For any rule $a \leftarrow t$ we can equate a with $t+a$, and we can unfold again this new occurrence of a with a rule $a \leftarrow t'$ to get $t + t' + a$, and so on. We can thus end up with a choice containing all the bodies of rules for a (possibly repeated).

6.5 State transition

We are now ready to define the possible state transitions that an agent may perform in its operation.

Definition 6.10 (State) *A* state *is a pair $\langle d, t\rangle$ of a database d and a task t.*

A transition between a current state and a new state occurs if there is a reduction between (a task congruent to) the current task and the new task through an enabled step. In this case the step updates the current database to the new database.

Definition 6.11 (State transition) *The* transition $\rightarrow_r$ *between states induced by a set of task rules r is the least relation satisfying the following rule.*

$$\frac{t \equiv_r t' \qquad t' \xrightarrow{\langle q,c\rangle} t''}{\langle d,t\rangle \rightarrow_r \langle d \triangleright c, t''\rangle} \quad \left(\begin{array}{l} d \vdash q \\ c \textit{ is consistent} \end{array}\right)$$

The null tasks resulting from the successful reduction of elementary tasks can be erased by the use of the congruence in a subsequent transition. A null task will only forcibly remain if the whole task is congruent with it, signifying the end of the activity of the agent.

Notice that reduction is only possible if an elementary task can be found. Allowing this is partly the purpose of the use of the congruence before applying a reduction. With it, named tasks can be unfolded any number of times until we reach elementary tasks. The congruence also handles the commutativity of the choice, allowing the choice reduction rule to pick any one of the summands for reduction (repetitions don't matter, since the rest of the choice is thrown away). The summands themselves may have been unfolded before applying the reduction, so that a named task may have been unfolded to a task containing an executable elementary task. The congruence also takes care of the commutativity of the parallel composition, so this executable elementary task may come from any member of a choice or a parallel composition. Since sequential composition is not commutative, the executable elementary tasks may only come from the left element of a sequential composition. As for the synchronous composition, its reduction rule is symmetrical by requiring elementary tasks to be available on both components, so there was no need to consider its commutativity in the congruence.

It is important to realize that task rules are *not* associated with reduction steps. This is in contrast to concurrent logic languages, where rule application is intrinsically tied to the notion of performing an execution step. In our case, rules unfold named tasks through the use of the congruence relation, but an arbitrary number of rules may be unfolded between steps.

We can also see that the initial task can be statically unfolded to an arbitrary depth, using the congruence, without affecting the operational semantics. We cannot, however, get totally rid of the dynamic use of the congruence, for three reasons: null tasks are dynamically generated and must be erased (this is particularly important in tasks of the form $\mathbf{0} \cdot t$), the use of commutativity cannot be statically decided, and recursive rules may require an infinite unfolding.

7 Introducing variables

We now refine the model to consider that query and command propositions, as well as named tasks, may have an inner structure with variables amenable to substitutions.

7.1 Variables

Assumption 7.1 (Variables) *There is a countable set $\mathcal{V}$ of* variables. *We shall use the letter v to range over variables.*

The introduction of variables raises several concerns: the definition of free variables and variable scoping, the role of variables in queries and commands, and the status of shared variables in tasks.

As we abstract from any particular syntax of propositions and named tasks, we have to make the following assumption.

Assumption 7.2 (Free variables) *The set $[x]$ of the* free variables *of x is defined for all propositions and named tasks.*

We impose an important restriction on databases.

Assumption 7.3 (Closed databases) *Databases contain no free variables.*

The reason behind this assumption is that the database can be globally accessed by any task, so if a free variable could be put there through a command it could then be transfered from its original scope in the task to another one, via a query, and this is clearly undesirable.

We are going to enforce that variables in tasks are just meant as placeholders for data communication among processes, and not as a synchronization device. This is again at odds with the situation in concurrent logic languages, where variables serve this dual purpose. We believe it is a better and more easily verifiable programming methodology to have variables purely as data placeholders, and keep as process coordination mechanisms those explicitly introduced by the task composition operators.

One of our design principles, in keeping with the idea of a data placeholder role for variables, is that a variable in a task will always have been substituted by data when the task finishes its execution. Either the variable acts as an output of data during the execution of the task, or it acts as input of data provided by the prior execution of another task with an occurrence of the same variable.

Free variables in a task are going to be classified as input or flexible. Input variables in a task will be those required to have been substituted by data before the task becomes executable, whereas flexible variables will be the remaining ones, that can be substituted before or during the execution of the task.

The sole source of substitutions for variables is the query action, whose execution is assumed to bind free variables to variable-free data. Free variables in a query will be classified as flexible, as they can remain unbound until the query is executed. Free variables in commands, on the other hand, will be classified as input, since the execution of the command is meant to update the database without introducing free variables.

The placement of a task as part of a larger task may lead to a disambiguation of the status of its flexible variables, through composition operators that may impose an input role. An output role will only be imposed by the variable scoping mechanism, but thereby also making the variable non-free, so there is no need to consider output variables in tasks.

7.2 Tasks

We now extend the tasks $\mathcal{T}$ to incorporate a variable scoping operator.

Definition 7.4 (Tasks) *The* tasks $\mathcal{T}$ *are redefined as the least set satisfying the rules in definition 5.1 plus the one below,*

$$\textit{variable scoping} \qquad v \;\mathsf{in}\; t \;\in\; \mathcal{T} \quad \textit{if} \quad v \notin [t]^i$$

where $[t]^i$ *represents the input variables of* t*.*

The idea of a variable scoping $v \;\mathsf{in}\; t$ is of course that any free occurrence of the variable v in task t is never going to be equated with a variable in another task. Since input variables of t will have to be previously bound to some data, they must occur in some other task, so v cannot be one of them. Notice that although generally v will occur free in t this is not required.

Definition 7.5 (Free, input, and flexible variables) *The* free *and* input variables *of tasks are defined as follows.*

$$
\begin{aligned}
[\mathbf{0}] &= \emptyset & [\mathbf{0}]^i &= \emptyset \\
[a] & \text{ assumed known} & [a]^i &= \emptyset \\
[?p] &= [p] \text{ assumed known} & [?p]^i &= \emptyset \\
[!p] &= [p] \text{ assumed known} & [!p]^i &= [p] \\
[t_1 \mid t_2] &= [t_1] \cup [t_2] & [t_1 \mid t_2]^i &= [t_1]^i \cup [t_2]^i \cup ([t_1] \cap [t_2]) \\
[t_1 \cdot t_2] &= [t_1] \cup [t_2] & [t_1 \cdot t_2]^i &= [t_1]^i \cup ([t_2]^i - [t_1]) \\
[t_1 + t_2] &= [t_1] \cup [t_2] & [t_1 + t_2]^i &= [t_1]^i \cup [t_2]^i \\
[t_1 \diamond t_2] &= [t_1] \cup [t_2] & [t_1 \diamond t_2]^i &= ([t_1]^i - [t_2]^f) \cup ([t_2]^i - [t_1]^f) \\
[v \text{ in } t] &= [t] - \{v\} & [v \text{ in } t]^i &= [t]^i
\end{aligned}
$$

The flexible *variables $[t]^f$ of a task t are its free variables that are not input, that is $[t]^f = [t] - [t]^i$. A task t is said to be* closed *if $[t] = \emptyset$, and* input-closed *if $[t]^i = \emptyset$.*

Variables in queries are flexible and those in commands are input, as explained before. Variables in named tasks are all considered to be flexible, as no hint about their role is assumed to be coded in the syntax. Our desire to have a simple dataflow model lead us to consider that there can be no competition among parallel tasks to bind a variable, thus explaining why shared variables in a parallel composition are required to be input. For the sequential composition, the classification of a shared variable is the one it has in the first task, as this task will always be executed before. As any one of the two components of a choice may be chosen for execution, being input in either of them requires a variable to be also input for the composition. Since queries and commands from synchronized tasks will be merged together, being flexible in one of them makes a variable flexible in the composition. With scoping, the scoped variable (which, remember, cannot be input) ceases to be free.

Regarding the rules, we have the following definition.

Definition 7.6 (Free variables in rules) *The free variables $[r]$ of a set of rules r are defined as follows.*

$$[r] = \bigcup_{a \leftarrow t \,\in\, r} [t] - [a]$$

A set of rules r is said to be closed *if $[r] = \emptyset$.*

In order to handle variable scoping we extend the structural congruence with α-conversion and scope enlargement. For defining α-conversion we make clear the concept of free occurrences of variables.

Definition 7.7 (Free occurrences of variables) *An occurrence of a variable v in an expression e is said to be* free *if $v \in [e']$ for any subexpression e' of e containing that same occurrence.*

The new congruence allows the pulling of the scoping operators to the outermost layer, possibly using α-conversion along the way to avoid capture of free variables.

Definition 7.8 (Structural congruence) *The* structural congruence $\equiv$ *on tasks is the least congruence satisfying the rules in definition 6.8 plus those that follow,*

$$\begin{aligned} v \ \mathsf{in}\ t &\equiv v' \ \mathsf{in}\ t[v'/v] && \textit{if} \quad v' \notin [t] \\ (v \ \mathsf{in}\ t) \otimes t' &\equiv v \ \mathsf{in}\ (t \otimes t') && \textit{if} \quad v \notin [t'] \\ t \otimes (v \ \mathsf{in}\ t') &\equiv v \ \mathsf{in}\ (t \otimes t') && \textit{if} \quad v \notin [t] \end{aligned}$$

where $t[v'/v]$ stands for the result of replacing every free occurrence of v in t by v', and $\otimes$ stands for any one of $|$, $\cdot$, $+$, or $\diamond$.

We also have to modify the congruence induced by rules, to handle the pattern matching between named tasks and rule heads. We consider that a named task can only be unfolded using a rule whose head is a strictly more general pattern, i.e. when there is a substitution for the head variables yielding the named task. This is in accordance with the approach taken in concurrent logic languages, but one must remember that the synchronization effect present there cannot be achieved here, because a flexible variable cannot be shared with a parallel task. Here, if an executable named task cannot match a rule head, it never will.

To redefine the congruence we must assume the existence of appropriate terms and substitutions. We remain at an abstract level regarding the actual syntactic structure of terms and named tasks.

Assumption 7.9 (Terms) *There is a syntactic universe $\mathcal{U}$ of* terms *that includes the variables $\mathcal{V}$. The letter u will be used to range over terms.*

Definition 7.10 (Substitution) *The family $\Sigma_{\wp^f(\mathcal{V})}$ yields, for each finite set of variables V, the set Σ_V of all functions $V \to \mathcal{U}$, called* substitutions *for V.*

Definition 7.11 (Application of substitutions) *The* application of a substitution $\sigma \in \Sigma_V$ *to a syntactic expression e, written $e\sigma$, is the result of re-*

placing each free occurrence of each variable $v \in V$ in e by $\sigma(v)$. This result is called an instance *of e.*

Assumption 7.12 (Substitution closure) *The sets of terms $\mathcal{U}$ and named tasks $\mathcal{A}$ are closed under the application of substitutions.*

Definition 7.13 (Ground terms and propositions) *The* ground *terms $\overline{\mathcal{U}}$ and the ground propositions are those containing no free variables. We say a substitution $\sigma \in \Sigma_V$* grounds *$V' \subseteq V$ if σ binds V' to ground terms, i.e. for all $v \in V'$ we have $\sigma(v) \in \overline{\mathcal{U}}$. A substitution $\sigma \in \Sigma_V$ is called* ground *if it grounds V. We represent by $\overline{\Sigma}_V$ the subset of ground substitutions of Σ_V.*

A fundamental idea of our computation model is that the database is always ground, and variables in tasks are meant to be substituted by ground terms coming from the database. We want to guarantee that variables in a task will have been made ground upon termination of the task.

We now redefine the congruence induced by task rules.

Definition 7.14 (Structural congruence) *The* structural congruence *$\equiv_r$ on tasks induced by a set of task rules r is the least extension of the structural congruence $\equiv$ in definition 7.8 that satisfies:*

$$a \equiv_r t\sigma + a \quad \textit{if} \quad \begin{cases} a' \leftarrow t \in r, \\ \sigma \in \Sigma_{[a']} \textit{ grounds } [a'] - [t]^f, \\ a = a'\sigma. \end{cases}$$

Let us analyse the requirement that σ grounds any head variable not occurring as flexible in the body. There are two possible situations for such a variable v. One is when v occurs as an input variable in the body. It must be grounded before execution, but given that variables in a are classified as flexible, a non-ground term matching v would have no guarantee of becoming ground before executing the body. The other situation is when v does not occur in the body. In this case we need groundness for maintaining our assumption that upon the successful execution of a it has been grounded. The only other possibility for a head variable is to occur as flexible in the body, and in this case the execution of the body will guarantee the grounding of any matching term from a, which is therefore not required to be ground. The constraints on σ provide yet another reason for the congruence to be used dynamically, since a named task's variables may have to be bound by some other prior task before the unfolding with a rule can take place.

In view of the requirement that input variables of a task must become ground before the task becomes executable, we constrain the definition of states.

Definition 7.15 (State) *A* state *$\langle d, t\rangle$ consists of a database d and an input-closed task t.*

We have seen that we can bring the scoping operators of a task to the outermost layer. We can then get rid of them altogether, given that any scoped variable attaining the outermost layer has been renamed apart from all other variables, and is never an input variable of the scoped task. This riddance of outermost scoping operators is done through an equivalence between states.

Definition 7.16 (State equivalence) *The* equivalence $\Leftrightarrow_r$ *between states induced by a closed set of rules r is the least equivalence relation satisfying the following rules.*

$$\begin{aligned} \langle d, t\rangle \Leftrightarrow_r \langle d, t'\rangle \quad & \textit{if} \quad t \equiv_r t' \\ \langle d, v \text{ in } t\rangle \Leftrightarrow_r \langle d, t\rangle & \end{aligned}$$

Let us now discuss the role of free variables in elementary tasks. Ground propositions form the basis for the database semantics.

Assumption 7.17 *Ground propositions denote situation types.*

We already assumed that databases are indeed ground, but what about query and command propositions? Free variables in queries are classified as flexible, so they may still be free when the query is ready for execution. A query with free variables is going to be interpreted as having a nondeterministic semantics given by the choice among all its ground instances. On the other hand, we took the stand of not wanting a nondeterministic semantics for commands. This explains the classification of free variables in commands as input, thereby requiring them to be grounded before the command is executed. The sole source of substitutions for flexible variables is the execution of queries, those substitutions being always ground.

We can now redefine the transition.

Definition 7.18 (State transition) *The* transition $\longrightarrow_r$ *between states induced by a closed set of rules r is the least relation satisfying the following rule.*

$$\frac{s \Leftrightarrow_r \langle d, t\rangle \qquad t \xrightarrow{\langle q, c\rangle} t'}{s \longrightarrow_r \langle d \triangleright c\sigma, t'\sigma\rangle} \quad \left(\begin{array}{l} \sigma \in \overline{\Sigma}_{[q]} \\ d \vdash q\sigma \\ c\sigma \textit{ is consistent} \end{array} \right)$$

Notice that task reduction retains its old definition, and that the new variable scoping operator is handled solely by state equivalence. We can show that state transition is well defined, i.e. that if r is a closed set of rules, s a state, and $s \longrightarrow_r s'$, then s' is a state; details are presented elsewhere. [14]

8 Multiple agents

The model described thus far relies on a single global database for information sharing and synchronization among parallel tasks, as well as a global set of rules for task refinement. We now extend the model to encompass the possibility of multiple agents, each being associated with a local database and a local set of task rules. Agents will be named and agent names will be appended to tasks. However, we will not consider local tasks in each agent but rather a system-wide task expression, this being far more expressive. [d]

8.1 Syntax

The management of unique names is essential, along with the dynamic creation of agents. We already have a renaming mechanism for handling the uniqueness of scoped variables, so it is natural to use scoped variables as agent names. However, we also want to be able to pass names around and store them in databases, and for this they must be ground terms. The solution lies in a new scoping mechanism linking a variable, a set of rules, a database, and a task. When such a scoped task is brought to the surface of the global task, the variable is substituted by a name.

Assumption 8.1 (Names) *The set of* names $\mathcal{N}$ *is a distinguished countable subset of the ground terms* $\overline{\mathcal{U}}$. *We shall use the letter* n *to range over names.*

Definition 8.2 (Tasks) *The* tasks $\mathcal{T}$ *are redefined as the least set satisfying the rules given in definition 7.4 plus the following ones,*

$$\begin{array}{rl} \textit{delegation} & b@u \;\in\; \mathcal{T} \\ \textit{agent scoping} & v\!:\!r\!:\!d \;\mathsf{in}\; t \;\in\; \mathcal{T} \end{array}$$

where b *is an atomic task in* $\mathcal{A} \cup \mathcal{Q} \cup \mathcal{C}$, *and* u *is a term in* $\mathcal{U}$. *Atomic tasks by themselves* $(a, ?p, !p)$ *are no longer part of the tasks.*

The idea behind an agent scoping task $v\!:\!r\!:\!d \;\mathsf{in}\; t$ is that v is used inside t to refer to an agent with rules r and database d, typically through subtasks of the form $t'@v$. Delegated tasks, as we will see, may be regarded operationally as remote procedure calls. An agent can delegate the execution of a task t to another agent n by launching the task $t@n$, whereby t is reduced (and likewise the resulting subtasks) using the rules and database of the agent n.

Although semantics will only be defined for delegated tasks of the form $b@n$, for n a name, we cannot syntactically enforce this, for names may appear in propositions and these may be stored and retrieved from databases (names may also appear in named tasks and be passed around).

The syntax for tasks only allows for delegation of atomic tasks. Delegation of generic tasks may be defined as a derived constructor.

[d] A fully distributed version can be implemented and formally described, but resorting to low-level communication mechanisms for programming protocols among agents.

Definition 8.3 (General delegation) *Delegation over generic tasks is a derived constructor, defined by induction on the structure of tasks as follows,*

$$\begin{array}{rlrl}
\mathbf{0}@u &= \mathbf{0} & (v \text{ in } t)@u &= v \text{ in } t@u \\
(t@u)@u' &= t@u & (v\!:\!r\!:\!d \text{ in } t)@u &= v\!:\!r@v\!:\!d \text{ in } t@u \\
(t \otimes t')@u &= t@u \otimes t'@u & r@u &= \{a \leftarrow t@u \mid a \leftarrow t \in r\}
\end{array}$$

where $v \neq u$.

Example 8.4 *Producer-consumer.*

The task below instantiates three agents: a producer of items of some kind, a consumer for those items, and a buffer used to communicate them. Notice how the producer puts an item in the buffer while producing the next item, and how it goes on for another round when these two actions are done.

$$\begin{array}{l}
\mathsf{buffer} : \cdots : \nabla \ \mathsf{in} \\
\mathsf{producer} : \mathsf{pp}(X) \leftarrow (\mathsf{put}(X)@\mathsf{buffer} \mid \mathsf{prod}(Y)) \cdot \mathsf{pp}(Y) \\
\qquad\qquad\quad \mathsf{prod}(X) \leftarrow \cdots : \nabla \ \mathsf{in} \\
\mathsf{consumer} : \mathsf{cp} \leftarrow \mathsf{get}(X)@\mathsf{buffer} \cdot \mathsf{cons}(X) \cdot \mathsf{cp} \\
\qquad\qquad\quad \mathsf{cons}(X) \leftarrow \cdots : \nabla \ \mathsf{in} \\
(\mathsf{prod}(X) \cdot \mathsf{pp}(X))@\mathsf{producer} \mid \mathsf{cp}@\mathsf{consumer}
\end{array}$$

□

Sets of rules may have free variables that are regarded as global parameters and must be substituted by ground terms before the rules are used. This justifies the following classification of variables in the new tasks.

Definition 8.5 (Free and input variables) *The* free *and* input variables *of delegation and agent scoping tasks are defined as follows.*

$$\begin{array}{rlrl}
[b@u] &= [b] \cup [u] & [b@u]^i &= [b]^i \cup [u] \\
[v\!:\!r\!:\!d \text{ in } t] &= ([r] \cup [t]) - \{v\} & [v\!:\!r\!:\!d \text{ in } t]^i &= ([r] \cup [t]^i) - \{v\}
\end{array}$$

8.2 Operational semantics

We have to extend the structural congruence on tasks. The main idea is to pull agent scoping to the outermost layer, similarly to what is done with variable scoping.

Definition 8.6 (Structural congruence) *The* structural congruence *is the least congruence on tasks satisfying the rules given in definition 7.8 plus those that follow.*

$$
\begin{aligned}
v\!:\!r\!:\!d \;\mathsf{in}\; t &\equiv v'\!:\!r[v'/v]\!:\!d \;\mathsf{in}\; t[v'/v] && \textit{if} \quad v' \notin [t] \cup [r] \\
t \otimes (v\!:\!r\!:\!d \;\mathsf{in}\; t') &\equiv v\!:\!r\!:\!d \;\mathsf{in}\; (t \otimes t') && \textit{if} \quad v \notin [t] \\
(v\!:\!r\!:\!d \;\mathsf{in}\; t) \otimes t' &\equiv v\!:\!r\!:\!d \;\mathsf{in}\; (t \otimes t') && \textit{if} \quad v \notin [t']
\end{aligned}
$$

We have also to redefine the congruence induced by rules. The basic idea is to regard a rule $a \leftarrow t$ within an agent n as a shorthand for $a@n \leftarrow t@n$. We will only be able to unfold tasks of the form $a@n$ where a is a named task and n is indeed a name, not a free variable or any other term, otherwise we couldn't guess which rules to use. But to define a global congruence we have to consider not a particular set of rules but rather the sets of rules associated with the existing agents. For this it is convenient to consider the notion of a split set of rules.

Definition 8.7 (Split set of rules) *A* split set of rules *is a total mapping $\mathcal{N} \to \wp(\mathcal{R})$ from names to sets of rules. We shall use the letter R to range over split rule sets. A split rule set R is* closed *if for any name n the set of rules $R(n)$ is closed.*

Definition 8.8 (Structural congruence) *The* structural congruence $\equiv_R$ *on tasks induced by a split rule set R is the least extension of the structural congruence $\equiv$ in definition 8.6 satisfying the following rule.*

$$
a@n \equiv_R t\sigma + a@n \quad \textit{if} \quad \begin{cases} a' \leftarrow t \in R(n), \\ \sigma \in \Sigma_{[a']} \textit{ grounds } [a'] - [t]^f, \\ a = a'\sigma. \end{cases}
$$

We already introduced the concept of a split rule set; we now do likewise for databases.

Definition 8.9 (Split database) *A* split database *is a total mapping $\mathcal{N} \to \mathcal{D}$ from names to databases. We shall use the letter D to range over split databases. The semantics of a database D is given by $[\![D]\!]\,(n) = [\![D(n)]\!]$, for all $n \in \mathcal{N}$.*

Analogously to what is done with variable scoping, we can get rid of outermost agent scoping, but two important differences emerge. First, when eliminating the scoping operator the agent variable must be substituted by a name. Second, this usage of a name must be recorded in the state to avoid future clashes of agent names. With scoped variables this problem does not arise, because they exist only in the state's task and while there provoke the renaming of other scoped variables whose scope is being pulled to the surface. Agent names, on the contrary, can be stored in databases for later retrieval, leaving no trace in the task of their existence, thus raising a problem of possible clashing with names for other agents introduced afterwards. This justifies incorporating in the state the information on the names already used.

Definition 8.10 (State) *A* state $\langle N, R, D, t\rangle$ *is composed of a finite set of names* N*, a closed split rule set* R*, a split database* D*, and an input-closed task* t*.*

The agent scoping elimination is handled through equivalence between states. This equivalence is no longer parametric on a set of rules, as these are now incorporated in the state.

Definition 8.11 (State equivalence) *The* state equivalence $\Leftrightarrow$ *is the least equivalence relation satisfying the following rules,*

$$
\begin{aligned}
\langle N, R, D, t\rangle &\Leftrightarrow \langle N, R, D, t'\rangle \quad \textit{if} \quad t \equiv_R t' \\
\langle N, R, D, v \;\mathsf{in}\; t\rangle &\Leftrightarrow \langle N, R, D, t\rangle \\
\langle N, R, D, v\!:\!r\!:\!d \;\mathsf{in}\; t\rangle &\Leftrightarrow \langle N \cup \{n\}, R + n \mapsto r[n/v], D + n \mapsto d, t[n/v]\rangle \\
& \qquad \textit{if} \quad \left\{ \begin{array}{l} [r] \subseteq \{v\}, \\ n \notin N. \end{array} \right.
\end{aligned}
$$

where $X + n \mapsto x$ *stands for the mapping* X' *differing from* X *only in that* $X'(n) = x$*.*

We can also lift our semantic domain to a version split across the set of names, while retaining the required formal properties.

Definition 8.12 (Split situation) *The* split situations $\mathcal{S}^*$ *are the total functions* $\mathcal{N} \to \mathcal{S}$ *from names to situations, equipped with a partial order* $\sqsubseteq$ *defined as the least relation satisfying* $s \sqsubseteq s'$ *if* $s(n) \sqsubseteq s'(n)$ *for every name* $n \in \mathcal{N}$*.*

We can easily show [14] that split situations are situation structures. For using the split situations as our new semantic domain, we must consider appropriate split propositions at the syntactic level.

Definition 8.13 (Split propositions) *The* split propositions *are the total functions* $\mathcal{N} \to \mathcal{P}$ *from names to propositions. The semantics of a split proposition* p *is given by* $[\![p]\!] = \{\, s \in \mathcal{S}^* \mid \forall\, n \in \mathcal{N}\; s(n) \in [\![p(n)]\!] \,\}$*.*

Being total mappings, split propositions are very abstract syntactic entities. To retain the old definition of task reduction it is convenient to consider a more concrete syntactic notion of proposition for queries and commands.

Definition 8.14 (@-propositions) *The set of* @-propositions *is the least set* $\mathcal{P}^*$ *containing all pairs* $p@n$ *of a proposition* p *and a name* n*. The semantics* $[\![\cdot]\!] : \mathcal{P}^* \to \mathcal{S}^*$ *is defined from the existing semantics* $[\![\cdot]\!] : \mathcal{P} \to \mathcal{S}$*, as follows.*

$$
[\![p@n]\!] = \{\, s \in \mathcal{S}^* \mid s(n) \in [\![p]\!] \,\} \qquad\qquad [\![p \wedge p']\!] = [\![p]\!] \cap [\![p']\!]
$$

We shall apply to @-propositions the same ranging letters we used for propositions.

We can guarantee the existence of a normal form for @-propositions consisting of split propositions; again check elsewhere [14] for details.

Assumption 8.15 (Normalization) *There is a semantics-preserving* normalization *function* $\langle\cdot\rangle : \mathcal{P}^* \to (\mathcal{N} \to \mathcal{P})$ *from @-propositions to split propositions.*

To conclude that our split entities are but a case of the general framework we need only to formalize the deduction relation and update function, by resorting to their versions for simple databases and propositions.

Definition 8.16 (Split deduction, split update) *The* deduction *relation* $\vdash$ *between split databases and @-propositions satisfies* $D \vdash p$ *if for every name* n *it is the case that* $D(n) \vdash \langle p\rangle(n)$. *The* update $D \triangleright p$ *of a split database* D *by an @-proposition* p *is the function satisfying, for every name* n, $(D \triangleright p)(n) = D(n) \triangleright \langle p\rangle(n)$.

We can verify that our deduction relation meets its purpose, by showing that it is sound and complete; we can also show that the syntactic update satisfies the intended connection to the semantic update. [14]

The new state transition relation is essentially similar to the previous one except for the extra components of the state.

Definition 8.17 (State transition) *The* transition $\longrightarrow$ *between states is the least relation satisfying the following rule.*

$$\frac{s \Leftrightarrow \langle N,\, R,\, D,\, t\rangle \qquad t \xrightarrow{\langle q,c\rangle} t'}{s \longrightarrow \langle N,\, R,\, D \triangleright c\sigma,\, t'\sigma\rangle} \quad \left(\begin{array}{l} \sigma \in \overline{\Sigma}_{[q]} \\ D \vdash q\sigma \\ c\sigma \textit{ is consistent} \end{array}\right)$$

Similarly to the case with a single agent, we can show that state transition is well defined. [14]

9 Further work

We now briefly outline some directions for further work in the *TAO* model.

9.1 Enhancements

TAO as presented here is already quite powerful, but some enhancements are still needed if we take seriously the aim of general applicability.

More general updates. The update operation is unnecessarily restrictive. It is based on enforcing the entailment of a situation, and this means that in some sense the "amount" of information never decreases; if some information disappears from the current situation it is only because it is replaced by a corresponding contradictory one. In order to have a general model of change we must account for update operations that actually remove information without replacing it. This can be achieved by considering that the update request is not captured by a situation but rather by a set of situations able to convey not only entailment but also disentailment requirements. Of course we have to restrict the nature of such sets in order to maintain the determinacy of updates. Notice that already the semantics of a command is considered to be a set of situations, but restricted to be an upward closed set with a least element. The solution lies in replacing the upward closure assumption by a less restrictive one.

Atomicity. We can use the synchronous composition to build an atomic action out of a combination of other actions, but only if these are already atomic. In general one may need an atomicity operator that will execute an arbitrary task as a single step from the point of view of parallel composition with other tasks. This is for example the concept behind database transactions.

Spawning. Of course a named task can be rewritten into a parallel composition of subtasks, that we can view as spawned processes, but termination of the original task is achieved only by the termination of all spawned subprocesses. It may be useful to introduce a notion of spawning that decouples the termination of spawner and spawned processes, i.e. an operation that immediately terminates but leaves as a (truly) side-effect an active task.

Modular specifications. Everybody appreciates the usefulness of a hierarchical structure of class definitions typical of object-oriented languages. Similar structuring principles can also be incorporated into *TAO*, but of course the search for a good semantic grounding is essential given the aim of using formal reasoning methods. Specially interesting is the possibility of going beyond static classification of agents and into dynamic modular composition. The ideas behind contextual logic programming[15] seem to be applicable to *TAO*.

Ambivalent syntax. Given the few assumptions on the syntax of terms we can imagine that task expressions or rules are themselves terms, and so can be passed around as data, and as such be constructed and deconstructed. This has obvious advantages in terms of the ability of *TAO* to model many practical phenomena such as mobile programs and their interpreters.

9.2 Semantics and verification

A lot needs to be done to fulfill the promise of *TAO* as a formalism for easing the task of verifying properties of programs.

For some safety properties we can develop a type system, for example capturing the notion that all delegated named tasks find a corresponding local rule. In general, though, we need a full-blown declarative semantics.

We aim at developing a denotational semantics which is fully abstract with respect to the relevant notion of observability that we need for the more general properties we want to prove, and a corresponding program logic.

Then, of course, practical reasoning tools need to be implemented for helping in performing proofs in the program logic.

Acknowledgements

This work was done under support from Esprit BRP 9102 "Coordination".

References

1. A. Porto and P. Rosado. The AbstrAct scheme for concurrent programming. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming*, volume 660 of *LNAI*, pages 216–241. Springer-Verlag, 1993.
2. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
3. Antonio Brogi and Paolo Ciancarini. The concurrent language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.
4. Jean-Pierre Banâtre and Daniel Le Métayer. The GAMMA model and its discipline of programming. *The Science of Computer Programming*, 15:55–77, 1990.
5. Jean-Marc Andreoli and Remo Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. In David H.D. Warren and Peter Szeredi, editors, *Logic Programming: Proceedings of the Seventh International Conference*, pages 495–510, Cambridge, MA, 1990. The MIT Press.
6. Vijay A. Saraswat. *Concurrent Constraint Programming*. Logic Programming Series. The MIT Press, Cambridge, MA, 1993.
7. Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

8. Robin Milner. *Communication and Concurrency.* C. A. R. Hoare Series Editor. Prentice Hall, 1989.
9. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992. Also as Tech. Rep. ECS-LFCS 89-85/86, University of Edinburgh.
10. Michel Chaudron and Edwin de Jong. Schedules for multiset transformer programs. In *this volume*.
11. Daniel Le Métayer. Higher-order multiset programming. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 18, pages 179–200. American Mathematical Society, 1994.
12. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
13. Frank de Boer, Catuscia Palamidessi, and Eike Best. Concurrent Constraint Programming with information removal. In Philippe Codognet, editor, *CCP'95: Proceedings of the 1995 International Workshop on Concurrent Constraint Programming*, Venice, 1995.
14. António Porto and Vasco T. Vasconcelos. Truth and action osmosis: the TAO computation model. Technical Report 95–1, Universidade Nova de Lisboa, 1995.
15. Luís Monteiro and António Porto. A language for contextual logic programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects.* MIT Press, 1993.