FORMAL SPECIFICATION OF DISCRETE SYSTEMS BY ACTIVE PREDICATES AND DYNAMIC CONSTRAINTS

Tommaso Bolognesi[1] and Andrea Bracciali[2]

Abstract - In this short paper we sketch an enhancement of our co-notation, a specification language for the behavioural description of a wide class of information processing systems which supports a constraint-oriented specification style. The enhancement substantially improves previous versions of the notation by allowing for the dynamic creation and elimination of constraints. The formal, operational semantics of the (extended) co-notation has been re-designed in a form which should favour both the exploration of further variants of the notation and its object-oriented implementation.

Introduction

The co-notation [1, 2] is a declarative specification language for the behavioural description of a wide class of information processing systems. Its wide applicability is due to the fact that the notation offers very few, but very powerful constructs, which are used for describing possibly nested constraints on the values of state variables and data structures, and on the values and the ordering in time of actions. We have presented elsewhere our notation, and have already illustrated the benefits of structuring behavioural descriptions in terms of compositions of constraints. (The reader interested in constraint-oriented specification may also refer to [3], where this style was first introduced in the context of the LOTOS language [4].) However, a non trivial limitation of our notation, in its original form, is that the set of constraints enforced by a co-spec (this term shall denote a specification in co-notation throughout the paper) is fixed. We want to relax this limitation for three reasons:

1. The behaviour of some of the systems that are conveniently described in a constraint-oriented style may well go through phases where further constraints are temporarily enforced, e.g. critical phases where the 'normal' operability has to be restricted by blocking some actions or narrowing the scope of their parameters.

[1] Istituto di Elaborazione dell'Informazione – C.N.R. – Pisa (t.bolognesi@iei.pi.cnr.it)
[2] Dipartimento di Informatica, Universita' di Pisa (braccia@di.unipi.it)

2. Although the co-notation has been conceived for supporting a constraint-oriented specification style, and constraints are usually understood as partial views on the system functionality, one can use them also for modelling concrete entities, or data-encapsulating objects, intended as in object-oriented design. In O-O design objects can be dynamically created and eliminated: we wish to do the same with the objects/constraints of the co-notation.

3. On a longer term, we are interested in the integration of constraint-oriented and object-oriented specification styles. In [5] Meyer writes that 'Object-Oriented design may be defined as a technique which, unlike classical (functional) design , bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs'. On the other hand, constraints are functional views. Thus, by reducing the conceptual distance between objects and constraints we attempt to bring back classical functional decomposition (in a revisited form) into software design, without giving up the well established advantages of O-O thinking. A preliminary investigation on this topic is found in [6], where a constraint-oriented enrichment of the ObjectZ specification notation [7] is described.

Another concrete objective of our work was to consider an alternative formulation of the formal semantics of the co-notation (originally given in [8]) which could naturally lend itself to the covering of the new dynamic features of the language.

An example of a dynamic constraint set

The purpose of this section is to illustrate, by an example, the new feature of constraint creation. A system called *agency* offers a database lookup service by dynamically creating search *agents.* Agents accept requests and money from users, access a data base, and return information to users, adding commercial information.

Agency

The agency interacts with the external world through the following actions: *Q*, for users to pose a lookup request, *A* for the agency to return to the users the requested information, *Db* for the agency to interrogate a database after user requests, and *Com* for the agency to obtain a commercial ad to insert in the answer to the users. The behaviours of the users, of the database and of the commercial ad provider are not part of the specification.

The agency maintains a set of *Known_Keys*, which represents the current information keys to be looked up on behalf of the users, and is initially empty. A *Safe* is also kept, for collecting money; its initial value is zero.

The agency simply consists in a number of independent agents that carry out the requested lookup work, and a mechanism for creating new agents when needed. Initially there are no agents.

```
agency [Q, A, Db, Com] :=
   state variables
      Known_Keys = {},
      Safe = 0
   constraints
      new_agent [Q!] (Q, A, Db, Com) <Known_Keys, Safe >.
```

The co-notation offers two types of constraint: *elementary* and *compound*. The fragment above is the definition of *compound* constraint *agency*, which is the top constraint of the specification. Its four actions are listed in square brackets after the constraint name, indicating the interface where the constraint is potentially active. The body of the constraint definition, following the definition symbol ':=', consists of two sections. Section *state variables* declares the two variables *Known_Keys* and *Safe* and their initial values. Section *constraints* includes the conjunction of constraint instantiations (there is only one here) that determine the initial behaviour of the system; this is called a *co-expression*. Action *Q*, in the instantiation of *new_agent*, is marked with the synchronisation decoration, which can in fact assume two forms:

1. closed action decoration: '!' -- by writing 'A!' in a constraint instantiation we indicate that the action does not synchronise with other instances of A supported by other constraint instantiations in the same co-expression.
2. open action decoration: '?' (the default case) -- by writing 'A?' or simply 'A' in a constraint instantiation we indicate that the action is required to synchronise with all the other open instances of A in the same co-expression.

Constraint *new_agent* represents the agent generation mechanism, which is triggered by *Q* actions. In its instantiation, actions *Q*, *A*, *Db*, and *Com* appear also in round parentheses: these parentheses are used here for passing action names to be used when creating new agent instances. Finally the constraint insists on the two variables *Known_Keys* and *Safe*, which are listed in angle brackets, and are call-by-reference parameters.

New_agent

A query from a user is posed to the agency via action *Q*. The action carries three values: *UserId* is the identity of the user, and shall be used also in the answer to the query; *Key* is the keyword about which the user requires information; *Fee* is the amount of money that the user spends for the requested service. When the *Key* is not one of those already known (*Known_Keys*) a new agent is created, who collects the user *Fee*, and records the *Key* and the *UserId* in order to satisfy the request.

```
new_agent [Val] (Q, A, Db, Com) <Known_Keys, Safe> :-
   Val = (UserId, Key, Fee),
   Key ∉ Known_Keys,
   Known_Keys' = Known_Keys ∪ {Key}
   create agent [Q!, A!, Db!, Com!]
                (Key, {UserId}, Fee) <Known_Keys, Safe>.
```

This fragment is an example of *elementary* constraint definition. Unlike compound constraints, elementary constraints (also called *active predicates*) can only insist on one action. The instance of active predicate *new_agent* in the body of the definition of compound constraint *agency*, insists on action *Q*, which is listed in square brackets. In the header of an active predicate definition the parameter in square brackets is always the reserved variable *Val* , which represents the value of that action. The header also contains a list of call-by-value parameters, in round parentheses, and a list of two state variables, *Known_Keys* and *Safe* upon which the constraint insists. The body of the definition, preceded by the symbol ':-', consists of a conjunction of logical predicates on that action value, on the pre-and post-values (relative to the action occurrence) of the state variables listed in the header, and on the other parameters. Post-values are identified by the primed decoration (e.g. *Known_Keys'*), as in the Z notation [9].

In static co-notation the set of composed constraints is fixed while in the dynamic extension it may increase or decrease. *Create* is a special predicate for doing this. It is always satisfied, it does not participate in the identification of satisfactory values for unbound variables, and its effect is to create an instance of a constraint, possibly parameterized by the variables known in its environment. In the nested constraint structure a newly created constraint appears one level up w.r.t. to its creation location. Hence, new agents pop up as brothers of constraint *new_agent*, in the body of *agency*.

Agent

An agent is in charge of looking up the database (action *Db*) for obtaining the information (*Info*) associated with a given *Key* and returning it via action *A* to all the users (*Ids*) that have expressed interest in that *Key*. The agent in charge of a *Key* can accept new queries only from users interested in that *Key*, and keeps track of this growing group of users. Users pay for this service; when the agent has collected enough requests, thus enough money in its *Budget*, it looks for the *Info* in the database, in turn paying for it out of its *Budget*, and also deposits the remaining money into the *Safe* of the agency. Then the agent delivers the information back to the interested users, enriched by a commercial ad that the agent obtaines from a separate source (*Com*).

```
agent [Q, A, Db, Com](Key, Ids0, Budget0)<Known_Keys,Safe> :=
   state variables
      Info = null,
      Ids = Ids0,
      Budget = Budget0
   constraints
      question    [Q]    (Key) <Info, Ids, Budget>,
      lookup      [Db]   (Key) <Info, Budget, Safe>,
      commercial  [Com]          <Info>,
      answer      [A]    (Key) <Info, Ids, Known_Keys>,
      sequencer   [Q, Db, Com, A].
```

Compound constraint *agent* insists all four actions of the specification. It is parameterized by a *Key*, by an initial set *Ids0* of users interested in that key, and by an initial value *Budget0* for the agent *Budget.* This constraint encapsulates three state variables: the *Info*, initially set to a null value, the set *Ids* identifying the interested users, and the *Budget*; the latter two state variables are initialized by the parameters *Ids0* and *Budget0.*

The co-expression after the keyword ‘constraints’ is a composition of four elementary constraints (*question, lookup, commercial, answer*), each one insisting on a different action, and one compound constraint (*sequencer*) determining the legal orderings of those actions. The execution of an action, say *A*, listed in the header of a compound constraint definition, requires the participation of all the constraint instantiations, in the definition body, which include '*A*?' (or '*A*') in their action lists, or the participation of just one constraint instantiation including '*A*!' in its action list For space reasons we illustrate only one of the five constraints above, namely the *Answer* active predicate. In co-notation one can also specify constraints that exclusively deal with the ordering of events: compound constraint *Sequencer* would be an example. Further discussion on event-ordering constraints and on the relation between the co-notation and Predicate/Transition nets [10] can be found in [1].

Answer

The agent returns the collected information (*Info*) to all the interested users, individually; then it removes its *Key* from the set of *Known_Keys*, and disappears.

```
answer[Val](Key)
      <Ids,Known_Keys>
:-
   |Ids| > 1,
   Val = (Id, Info),
   Id in Ids,
   Ids' = Ids - {Id}.
```

```
answer[Val](Key)
      <Ids,Known_Keys>
:-
   Ids = {Id},
   Val = (Id, Info),
   Ids' = {},
   Known_Keys'
      = Known_Keys - {Key},
   destroy agent.
```

Note that this active predicate has two alternative bodies – a circumstance that can be observed also in logic programming languages such as Prolog [11]. In particular, the second body describes the answer to the last user left, and takes care of removing the agent *Key* from the set of *Known_Keys*, and eliminating the agent itself. In general when a predicate has multiple definitions, anyone of them can be considered in order to satisfy its instantiation: when several bodies can be satisfied the choice is nondeterministic.

Aspects of the semantics

A co-spec is formed by a set of compound and elementary constraint definitions, called 'constraint definition environment' (denoted by Δ), and by an instantiation of a top (or 'main') compound constraint (denoted $Inst_0$). This instantiation is *closed*, meaning that its action parameters and value parameters are constants, and that there are no external state variables.

A co-spec may use standard data types (e.g. integers) and functions (e.g. sum), and may also include new definitions of such items. All these elements are used for expressing values and for building the bodies of elementary constraint definitions, but we do not need to be too specific about their syntax. What we basically use in those bodies is first-order logic with equality.

We have devised an operational, interleaving semantics for the co-notation. The semantics of a closed co-spec is a possibly infinite labelled transition system (LTS) in which transitions have the form τ---*(A, Val)*--->τ', where τ is a configuration tree fully describing the system state, *A* is an action name, *Val* is the value taken by the action, and τ' is the new configuration tree reached by the system after the action execution. A configuration tree is obtained by expanding a co-spec: each internal node represents a compound constraint instantation, and holds the current values of some state variables, while each leaf represents an elementary constraint instantiation. A configuration tree represents the hierarchy of constraint instantiations of a specification. The semantics makes use of a function called ***instantiation expander***, which, given a constraint instantiation, produces a configuration tree; the initial configuration tree for a co-spec with $Inst_0$ as top constraint instantiation shall be obtained by applying the expansion function to $Inst_0$. Finally, rules are provided for deriving labelled transitions from this initial configuration tree. For space reasons, in the sequel we can only provide summarized definitions of these semantic components. In our summary we emphasize the handling of state variables, while omitting the description of the constraint-synchronisation mechanism (this is the simple criterion for selecting different sets of active predicates supporting, in turns, different action executions, which is based on the inspection of the decorated action lists associated with the constraint instantiations of co-expressions.)

Configuration trees

A configuration tree is a finite tree structure that completely describes the state of the specified system; it contains the set of constraints currently in effect, and the current values of the state variables. We use the acronyms CC for compound constraint and EC for elementary constraint. A configuration tree is formed by two types of node: internal nodes correspond to CC instantiations, and are called CC nodes; leaves correspond to EC instantiations, and are called EC nodes. A CC node corresponding to the instantiation of a compound constraint *B* is a triple $(B, \underline{T}^h * \underline{TT}^h, \underline{S}^m * \underline{Val}^m)$ where $\underline{T}^h$, $\underline{TT}^h$ and $\underline{S}^m$ denote tuples of state variable names, $\underline{Val}^m$ denotes a tuple of values, the superscripts indicate tuple sizes, and '*' is the ordered, element-wise pairing of the elements of two tuples of equal size. $\underline{T}^h * \underline{TT}^h$ denotes the mapping between formal and actual external state variables of a constraint instantiation, and $\underline{S}^m * \underline{Val}^m$ is the assignment of values to its internal state variables.

An EC node corresponding to the instantiation of an elementary constraint (active predicate) *P* is a tuple $(P, \underline{T}^h * \underline{TT}^h, \pi,$ create_destroy_statement) where $\underline{T}^h * \underline{TT}^h$ is as in CC nodes, π is the logical body of the active predicate, expressed in terms of the (formal) external state variables of the node, possibly in primed form, and create_destroy_statement , which may be missing, is the statement possibly found at the end of the predicate body for creating or destroying constraints on-the-fly, upon action occurrences.

Constraint instantiation expander (τ_Δ)

This function takes a constraint instantation and returns a configuration tree for it, based on the constraint definition environment Δ. Its recursive definition has two clauses, one for CC and one for EC instantiations. Consider a co-spec with top CC instantiation $Inst_0$ and constraint definition environment Δ. The configuration tree associated with it shall be $\tau_\Delta(Inst_0)$. The configuration tree of a co-spec contains the values of the internal state variables that form the global system state, distributed in the internal nodes as variable-value pairs of the $\underline{S}^m$ * $\underline{Val}^m$ products. Indirect references to these values are found in the bodies of the active predicates at the leaves of the configuration tree; dereferencing is achieved by following a path upwards in the tree, and looking up the tables of internal or external state variables until a constant value is found.

Transition rules for configuration trees

These are inference rules for transitions of type $<\sigma, \underline{\tau}>$---$(A, Val)$---$>$ $<\sigma', \underline{\tau}'>$, where σ and σ' are assignments to state variables, and $\underline{\tau}$ and $\underline{\tau}'$ are configuration forests (sets of configuration trees). This general form of transition is useful for a compositional semantic definition, where transitions of a tree are provided in terms of the transitions of its children; simple transitions of type $\underline{\tau}$---(A, Val)---$>$ $\underline{\tau}'$, are a special case, corresponding to a complete, closed co-spec (for which no assignment to external state variables is necessary, since these are missing). We need two groups of rules. The EC transition rules define the transitions of EC nodes, which are the leaves of the configuration trees. The CC transition rules define the transitions of a configuration tree rooted at a CC node, based, recursively, on the transitions of its immediate subtrees. We use a trick: the set of names of constraints to be destroyed is recorded as a label of the transition arrow, and propagated up the derivation tree until a node corresponding to a constraint to be eliminated is actually found. At this point the subtree rooted at that node is removed, and its name is removed from the set in the arrow label.

As for the dynamic creation of new constraints, the semantics, expectedly, takes advantage of the already mentioned function τ_Δ.

Conclusions

The main purpose of the work partially described here has been one of devising an operational formal semantics for a constraint-oriented specification notation (co-notation) also supporting the dynamic creation and elimination of constraints. Constraints can also be understood as resources (like the 'agents' of our example) or *objects*, and this identification appears to us as a first step in the attractive direction of integrating constraint-oriented reasoning and Object-Oriented design.

A limitation of the dynamic co-notation is that the action lists and their decorations are defined statically; the dynamically created constraints cannot introduce new action names, or modify existing action decorations, but can only fit into the predefined synchronisation patterns.

We claim that the step from the current formulation of the semantics to the object-oriented implementation of an interpreter of the notation must be very short. We expect that the development of such a tool will shed more light on the relations between object-oriented and constraint-oriented specification styles, and on the feasibility of their integration.

References

[1] T. Bolognesi, 'Expressive Flexibility of a Constraint-oriented Notation', The Computer Journal, Vol. 40, No. 5, Oxford Univ. Press, 1997.

[2] T. Bolognesi, F. Accordino, 'A layer on top of Prolog for composing behavioural constraints', to appear in: Software Practice & Experience, John Wiley and Sons, 1998.

[3] C. A. Vissers, G. Scollo, M. Van Sinderen, 'Architecture and specification style in formal descriptions of distributed systems', in S. Aggarwal and K. Sabnani editors, Protocol Specification, Testing, and Verification VIII, North-Holland, 1988, pp. 189-204.

[4] E.Brinksma (ed.), ISO - Information Processing Systems - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour, ISO 4, February 1989, ISO, Geneva.

[5] B. Meyer, 'Reusability - the case for object-oriented design', IEEE Software, 4(2):50-64, 1987.

[6] T. Bolognesi, J. Derrick, 'A constraint-oriented style for object-oriented formal specification', to appear in IEE Proceedings - Software Engineering, 1998.

[7] R. Duke, G. Rose, G. Smith, 'Object-Z: a specification language advocated for the description of standards', Computer Standards and Interfaces, 17: 511-533, Sept. 1995.

[8] T. Bolognesi, F. Accordino, 'Constraint-oriented Specification Style and Notation', Technical Report B4-43-12-96, CNR - IEI, Pisa, 1996.

[9] Spivey, M. (1990)The Z Notation - A Reference Manual. Prentice Hall.

[10] H. J. Genrich, 'Predicate Transition Nets', Lecture Notes in Computer Science, 254, Springer-Verlag, 1987.

[11] L. Sterling, E. Shapiro, The Art of Prolog, The MIT Press, 1986.