From Modules to Structured Theories and Logic Objects: a Common "Program-based" Run-time Support

Enrico Denti, Antonio Natali and Andrea Omicini
DEIS – Università di Bologna
viale Risorgimento, 2
40136 Bologna - Italy
{enrico, natali, andrea}@deis33.cineca.it

Abstract

This work presents the main features of CSM, a contextual logic programming system (fully compatible with the SICStus Prolog programming environment) which aims at supporting evolutive software development methodologies in logic programming. Rather than fixing a set of policies, CSM provides mechanisms for the (dynamic) configuration of structured theories, handled as first class objects. CSM mechanisms allow code reusability and sharing, and provide binding rules fully supporting object-oriented programming techniques, without undermining logic programming declarativeness. The implementation is a peculiar CSM aspect, since contexts are represented by descriptors (similar to SICStus module descriptors), configured at context creation time, with no modification at abstract machine level. While this choice implies relatively high costs for context creation, it prevents any semantics distortion or computational overhead with respect to standard (SICStus) Prolog programs, and allows users to exploit open, evolving software components at a cost comparable with that of more static approaches to program composition.

1 Introduction

The modern approach to software design, development and maintenance, demands new models of application building where the traditional notion of phase-refinement is replaced by that of software evolution [4]. A well-known example of evolutionary system development is prototyping, whose effectiveness crucially depends on the possibility of changing prototype behavior with less effort than would be required to modify the production software.

When traditional (imperative) languages are used to build the final product, higher-level languages and special-purpose tools are required in order to deal with prototypes. In such a case, a prototype is usually intended to be only a partial representation of a system that can be used as an aid to analysis and design, rather than as a production tool.

Such a reductive view of prototyping seems to be intrinsically overcome when applications fit the area of logic programming, since logic languages can greatly reduce the distance between specification and coding as well as between prototype description and system implementation. However, in order to fully support software development techniques based on software evolution, logic languages have to be enhanced with suitable constructs for programming-in-the-large.

Among the proposals to extend logic programming towards modularity and object-oriented software modeling, contextual programming [8] aims at introducing a set of general purpose abstractions and mechanisms rather than prefixed knowledge structuring models and policies.

A context is defined as the composition of a set of theories (also called modules or units), each identified by a name and constituted by a collection of Horn clauses. Since contexts can be specified in a static way or dynamically created through extension goals such as u>>g, the contextual programming model provides a uniform approach to both static and dynamic knowledge composition and object-oriented programming. In fact, the extension goal can be interpreted as a request to execute a method (goal g) in the object obtained by pushing unit u on top of the object constituted by the current context. Moreover, when they are interpreted as structured binding environments, contexts can support a number of different binding policies, each leading to a different component architecture. Using the terminology reported in [7], local or eager binding mechanisms lead to conservative, closed worlds (where each structured component is self-contained, i.e. its behavior does not depend on external knowledge) while lazy binding supports evolving, open worlds (where the component behavior can change when specialized by new units).

While these mechanisms have already been proved to be powerful and general enough to provide a unifying support to the notions of object, inheritance, viewpoints, etc. (see [7] or [1]), how to give them an effective implementation is still an open problem. In order to give contextual logic programming the power of a fully developed programming environment (including an optimizing compiler, an efficient run-time support and all the classic programming tools like debuggers, graphical interface, etc.), the simplest strategy is to integrate contextual mechanisms into a widely-used and well-supported logic programming system. However, even in this case different choices are possible, some of which have been compared in [3]:

- contexts can be implemented through a meta-interpreter
- contextual programs can be translated into conventional Prolog programs
- contexts can be supported by extending the underlying (Warren) abstract machine

In this work we will discuss in detail another implementation approach, called program-representation-based, inspired by the SICStus Prolog programming environment [11], and also been advocated for SEPIA [9]. This version of contextual programming has been called CSM, i.e. Contexts like SICStus Modules, since the representation of a context is quite similar to that of a SICStus Prolog module. As a consequence, each context is a first-class object, identified by a unique name, which can work as a structured logic theory as well as a conventional object that can be part of different prototypes. A context database is maintained by CSM, and can be exploited by higher-level tools in order to assist the programmer in building and executing applications.

While other proposals (e.g. that reported in [6], [2]) aim at finding a good balance between expressive power and efficiency, CSM aims at fully supporting the highly dynamic nature of contextual mechanisms. The notion of theory is exploited both at abstract and implementation level to support conventional object-oriented concepts such as inheritance, classes, fixed or mutable instances with private or shared state. Besides, more advanced concepts like hypothetical reasoning and dynamic object (re)classification can be expressed.

The work is organized as follows. In section 2 we briefly recall the main concepts and mechanisms related to contextual programming and their impact on software design and development. Section 3 is devoted to the presentation of the main characteristics of the CSM implementation. Conclusions and further remarks are reported in section 4.

2 Contexts as Structured Software Components

In CSM, a context is a first-class structured object identified by an atomic name. A context `c` is recursively defined as a pair `(u,c1)`, where `u` is the name of the top unit of `c`, and `c1` identifies the first subcontext of `c`. The atom `emptyCtx` is used to denote the empty context, given as a constant. Since a unit is a named collection of (Prolog) Horn clauses, a context can be represented by the ordered list of its composing unit names, e.g. $[u_n,\ldots,u_1]$, where u_n is the first (most recent) unit of the context. [1]

The constructor `createCtx(U,C1,C)` builds a new context by extending the context referred to by the logic variable `C1` with the unit referred to by `U`, and links the unbound variable `C` to the new context identifier. Two selectors `topUnit/2` and `sub1Ctx/2` are introduced so that the goal `createCtx(U,C1,C), topUnit(C,U), sub1Ctx(C,C1)` is true when `U` and `C1` are properly instantiated.

The built-in predicate `C<-G` , where `C` is a context and `G` a goal, performs a switch of the current computation (line of reasoning) to context `C`, solving goal `G` within the structured theory `C`. In an object-oriented perspective, the context-switch operator sends message `G` to object `C` whose methods are the clauses made visible by `C`.

Context extension operators `>:>/2` and `>>>/2` (for cactus and linear extension, respectively, as mentioned in [1]), allow users to implicitly create and use objects in a dynamic way.

CSM supports two kinds of non-local calls (as defined in [7]) and two corresponding binding policies: eager calls and lazy calls. Let us consider a predicate call `p/n` within a unit u_i in the current context $[u_n,\ldots,u_i,\ldots,u_1]$. If a predicate `p/n` is eager-called within u_i, a definition for `p/n` is sought through the subcontext $[u_i,\ldots,u_1]$, called the current bindcontext. On the other hand, when a predicate call `q/n` is lazy-called within a unit u_i, a definition for `q/n` is sought through the whole current context.

In order to support non-local binding policies, CSM keeps track of both the current context and the current bindcontext, and provides two primitive operators: `curCtx(C)`, which binds `C` to the current working context, and `curBindctx(C)`, which binds `C` to the current bindcontext.

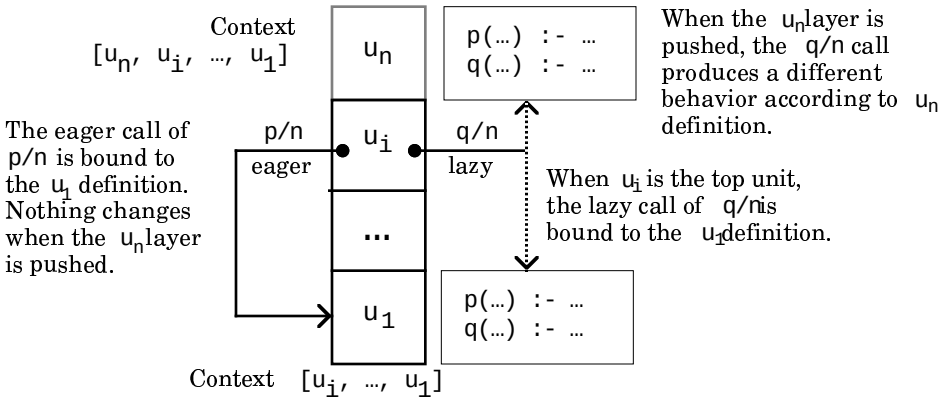

Figure 2.1. Eager and lazy binding policies.

CSM binding policies allow users to structure components in different ways. In particular, eager binding leads to conservative, closed architectures, while lazy binding supports evolving, open architectures. As shown in figure 2.1, the definition of `p/n` in u_i depends only on the

---

[1] For instance, using list notation, context `[u2,u1]` is defined as the pair `(u2,[u1])`, where `u2` is its top unit and `[u1]` its first subcontext. Since the empty list `[]` is used for the empty context, `(u1,[])` denotes context `[u1]`.

previous layers $u_{i-1},\ldots,u_1$. If every other call in u_i is local or eager, the behavior of the component represented by the context $[u_{i-1},\ldots,u_1]$ can be completely defined when the context is created. Even if a more specialized component is built, by pushing new layers upon u_i, the behavior of the sub-component $[u_{i-1},\ldots,u_1]$ remains unchanged. Thus, eager binding cannot support the incremental programming methodology of the OO paradigm, where the behavior of class methods depends on specific instance knowledge. On the other hand, lazy binding allows a unit to work as an open, evolving component, with no need to rewrite it in order to enhance or extend its features. In fact, lazy binding is the mechanism required to support the notion of self in OOP. In figure 2.1, since `q/n` is lazy-called, the behavior of the sub-component $[u_i,\ldots,u_1]$ changes when the upper layer u_n is added, since a new definition for `q/n` is found there.

To express local, eager and lazy calls, CSM adopts the following syntactic conventions. A goal `g` called in a unit `u` is treated as a local call if there is a clause for it in `u` (i.e., when `functor(g)`$\in \upsilon_d$, where u_d is the set of local predicates explicitly defined within unit `u`). Otherwise, (i.e. when `functor(g)`$\notin \upsilon_d$) the call is considered to be an implicit request for an eager call, and eager binding policy is adopted. Lazy calls have to be explicitly expressed by prefixing the # (or the equivalent `self`) operator to a goal call. Moreover, an eager call can be explicitly forced through the `super` prefix operator. Let us consider the following example.

Example 2.1.

```
:- unit(carOwner).          :- unit(pcOwner).
owns(antonio,alfa33).       owns(antonio,lemon).
owns(evelina,volvoPolar).   owns(enrico,ibicomp).
owns(paola,fiat500).        owns(paola,macFX).
hasAcar(X) :- owns(X,_).    hasApc(X) :- owns(X,_).

:- unit(hasA).              :- unit(hasMore).
hasA(X) :- #hasAcar(X).     hasBoth(X) :- hasAcar(X),
hasA(X) :- #hasApc(X).                    hasApc(X).
```

A possible world for car owners is the context `[carOwner,hasA]`, which is built as a consequence of the following query:

```
?- createCtx(hasA,emptyCtx,HasCtx),
   createCtx(carOwner,HasCtx,CarOwnerCtx),
   CarOwnerCtx <- hasA(X).
```

or of the simpler and equivalent query

```
?- [carOwner,hasA] <- hasA(X).
```

The above queries would return:

```
X = antonio; X = evelina; X = paola;
no
```

In fact, a binding for `hasA/1` in `[carOwner,hasA]` is found in unit `hasA`, where only the lazy call of `hasAcar/1` succeeds in finding the `carOwner` clause; then, the `owns/2` call is treated as a local call and returns the above computed answers. While unit `carOwner` works as a closed

theory, unit `hasA` is an open theory whose behavior is determined by the current context: in fact, all its calls are non-local. Furthermore, as they are lazy-calls, evolutive computations may occur. For instance, if a goal

```
pcOwner >>> hasA(X)
```

is called when `[carOwner,hasA]` is the current context, and `X` is unbound, we get:

```
X = antonio; X = evelina; X = paola;
X = antonio; X = enrico; X = paola;
no
```

Due to its "evolutive" implementation, the same goal `hasA(X)` now gives different answers, since it is now solved in the extended context `[pcOwner,carOwner,hasA]`.

On the contrary, unit `hasMore` is still an open theory (since it uses two undefined predicates, therefore interpreted as two eager calls), but is conservative, since its behavior is completely determined by the previous layers (subcontexts) in the context. For instance, if we try to solve the following goal in the context `[pcOwner,carOwner,hasA]`, with `X` unbound:

```
hasMore >>> hasBoth(X)
```

we should obtain:

```
X = antonio; X = paola;
no
```

Now, any other try to further extend the obtained context `[hasMore,pcOwner,carOwner,hasA]` with any other unit will not modify the situation: `hasBoth/1` in `hasMore` will always return the same answers as long as the sublayer (its subcontexts) remains unchanged.

❑

Expressive power of contexts

Contextual mechanisms provide a high modularity degree for knowledge representation and control, together with the capability of dynamically altering software system behavior and architecture with no permanent side-effects on already working components. Modularity is directly captured by the concept of unit. Units composing a context can be interpreted as an explicit representation of an inheritance tree branch: thus, contexts can be interpreted as object hierarchies, where both static (the concept of super) and dynamic (the concept of self) inheritance can be exploited. As a result, contexts provide a modeling power similar to that of object-oriented languages. Besides integrating both static and dynamic configuration of structured objects in a unique framework, they promote a declarative style of programming, with uniformity of properties (no distinction between methods and state) and referential transparency in method calling. Dynamic context extension can be used to perform run-time specialization of statically-built software components.

The following example aims at illustrating the above considerations.

Example 2.2.

```
:- unit(animal).                :- unit(elephant).
move(walk)  :- #legs(_).        legs(4).
move(run)   :- #legs(2).        ears(2).
move(gallop):- #legs(4).        trunk(1).
move(fly)   :- #wings(2).

:- unit(fred).                  :- unit(bird).
legs(3).                        legs(2).
                                wings(2).
```

An elephant object can be represented by context `[elephant,animal]`, which expresses the fact that an elephant can be seen as a specialization of the concept of animal.[2] A goal like

```
?- [elephant,animal] <- move(X)
```

would produce the following computed answers:

```
X = walk; X = gallop;
no
```

We could say that the `elephant` class statically inherits `move/1` methods from its parent class `animal` which, for its part, dynamically inherits the number of `legs` from its subclass.

If `fred` is a three-legged elephant, the goal

```
?- [fred,elephant,animal] <- move(X)
```

yields, as expected, the answer

```
X = walk;
no
```

since the "self"-binding of `animal:legs/1` now involves unit `fred`.

The same result can be achieved by performing a linear context extension operation:

```
fred >>> move(X)
```

while working in the "elephant context" `[elephant,animal]`. Such an operation may be seen as a dynamic specialization of the elephant class, in the sense of `fred`'s three-legged being.

❑

CSM mechanisms allow users to extend the scope to the field of hypothetical reasoning and dynamic object classification. For instance, if we do not know whether `fred` is an elephant lacking a leg or a bird with an extra leg, we can try to extend both classes (`[elephant,animal]` and `[bird,animal]`) with `fred`, and then reason on the consequences of the two cases. .

---

[2] Proper CSM declarations, such as `unit elephant inherits animal`, also allow static taxonomies to be created.

However, while contexts can suitably model the notion of class and instance, they lack the ability to capture the concept of a uniquely identifiable object with an internal modifiable state. In fact, in the form discussed here, contexts can support functional objects only. An extension of CSM to express objects with evolving state is presented in [10].

3 The Implementation

To be effective, contextual mechanisms have to be implemented efficiently. But efficiency is not enough. In order to put contexts in an effective software engineering perspective, contextual mechanisms should be integrated in an efficient, widely-used and well-supported system. To achieve this goal, we selected the SICStus Prolog programming system. Moreover, we decided that the following requirements had to be met:

- the implementation of contexts must introduce neither semantics distortion nor computational overhead with respect to standard (SICStus) Prolog programs;
- contexts should be first-class objects which can be referred to by logic variables;
- contexts should be memo-ized in an internal system knowledge base and reused when needed.

Given these requirements, the challenging issue was to extend an already existing system so as to obtain a good degree of efficiency with minimal intrusion. In order to achieve this we explored the space of the implementation techniques depicted in the tree of Figure 3.1:

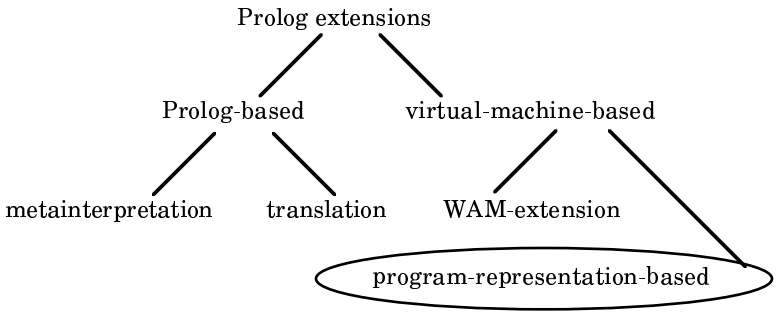

Figure 3.1. A taxonomy of Prolog extension techniques.

Translation and metainterpretation are absolutely non-intrusive techniques, since any Prolog-based approach adds a Prolog layer between the user and the underlying Prolog machine, leaving the machine itself untouched. In particular, translation transforms a contextual program P into a pure Prolog program P', semantically equivalent to P. With metainterpretation the Prolog interpretation cycle, extended with the new linguistic features, is reproduced by a metainterpreter, handling contextual programs as its data.

The main drawback of any Prolog-based approach is that it is affected by quite a low degree of efficiency. As illustrated in [3] (where some comparison figures are reported), a translator always works better than a metainterpreter, but it is affected by an undesirable performance decrease when program complexity grows.

The virtual-machine-based approach classically addresses the efficiency issue by modifying the language run-time support. In particular, the WAM-extension technique consists of enhancing the abstract Prolog machine in order to directly support new language features by introducing a set of new instructions and structures (a possible solution for contexts

is reported in [6]). This technique has two main drawbacks (for a more complete discussion see [9]). The first is that the WAM emulator code is highly optimized, i.e. highly intricate and tricky. The second is that the WAM extension suffers from the overhead it imposes on all computations, including standard Prolog ones.

However, the particular structure of the SICStus machine emulator suggests a specialized version of the virtual-machine-based approach, directly based on the representation of Prolog programs. In figure 3.1 such an implementation approach has been called program-representation-based. In fact, the SICS-WAM relies on a program representation structured as a tree, rather than on a plain sequence of WAM instructions. In Figure 3.2 we depict the tree-based program representation as a separate block handled by a specific subsystem (the program representation manager).

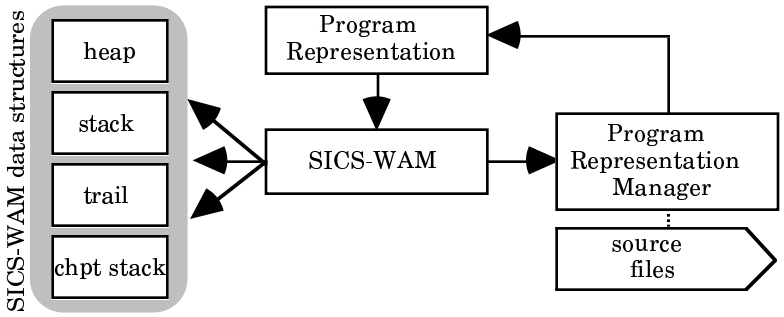

Figure 3.2. A SICStus Prolog architecture view

The root of the program tree is a hash table which leads to module descriptors. Each module has a name and a separate predicate space, represented again by a hash table pointing to predicate descriptors. These are the tree leaves, each one pointing to its code; for its part, the predicate code can be structured into several interconnected chains of clause codes (as in the case of compiled code). This structure is motivated essentially by the need to support some non-standard features, such as:

- a flat module system with no change to the basic WAM structure;
- a more efficient clause indexing scheme (avoiding `try`, `trust` and other indexing WAM instructions);
- incremental loading of clauses and dynamic predicates (for clause assertion and retraction);
- coexistence of compiled, interpreted and C-written Prolog predicates.

CSM architecture

As the SICS-WAM can access clause codes at several different levels of the tree, an indirect addressing mechanism is provided, even if compiler smartness forces almost-direct access in most frequent (and useful) cases. This configuration allowed us to meet two fundamental requirements for context implementation:

1) a global naming mechanism, associated to a partition of the clause space in order to implement the unit abstraction;
2) an indirect addressing mechanism, since contexts need dynamic binding.

The first requirement was satisfied by simply mapping CSM units onto SICStus modules, so as to reuse SICStus module names and separate predicate spaces. In particular, binding of local calls may consequently be delegated to SICStus mechanisms so that CSM intra unit computations are handled by SICStus as normal Prolog computations.

As far as the second point is concerned, our first choice was also to represent Contexts as SICStus Modules (whence CSM) so as to use the SICStus indirect addressing scheme for context binding, and to have contexts (for free) represented as physical objects in the system memory. On the other hand, this seemed to contrast with our "minimal intrusion" requirement: in fact, a frequent operation like context creation easily led to an undesirable crowding of SICStus module space; moreover, the need to record and handle specific CSM information forced us to extend SICStus native structures. Then, we finally switched to a configuration of the CSM Knowledge Base which uses SICStus unit representation in terms of modules as its underlying layer, and superimposes all CSM-specific information.

A context is then implemented like a SICStus module in the CSM KB space. That is, similarly to SICStus modules, a context is represented by a context descriptor, with its own identifier and predicate space. The descriptor of a context c constitutes the concrete representation of an abstract binding set c_d associated with c; c_d can be formally defined as follows:

$$\mathit{emptyCtx}_d ::= \varnothing \qquad \{\text{apart from Prolog built-in predicates}\}$$

$$c_d ::= \{(u,p/n) \mid p/n \in u_V\} \cup \{(m,p/n) \mid (m,p/n) \in c1_d, p/n \notin u_V\}$$
$$\{\text{if } \mathit{topUnit}(c,u), \mathit{sub1Ctx}(c,c1) \text{ holds}\}$$

where u_V is the set of predicates (implicitly or explicitly) declared as visible in u. In other words, c_d consists of those predicates declared as visible in the top unit u of c, and of those predicates made visible by the first subcontext of c, unless redefined by u.

The context predicate set is represented via a hash table, working as a binding environment descriptor. Thus, context creation (which constitutes the CSM main computational cost when performed at run time[3]) consists essentially of a suitable (hash) table configuration. First, a new copy of the first-subcontext table is allocated, then the top-unit visible predicates are introduced. CSM default predicate-overriding semantics is achieved by overwriting the first-subcontext predicate entries with the top unit ones when collisions occur.

Contexts are permanently recorded in the CSM KB; they are deallocated only when a component unit is redefined. Thus, context creation needs to be performed only once: when a context has to be used again during a computation (for instance, after backtracking), it is simply restored from the CSM KB through the context table.

Context predicate tables are used to perform non-local binding efficiently. When a non-local call (eager or lazy) takes place in a context, a hash access to the context predicate table is performed in order to find (if a suitable predicate exists) the proper predicate information, similar to the SICStus module call mechanism. All information needed (the owner unit and the corresponding bindcontext, the code address, etc.) can then be accessed at once, so that the code can be immediately executed after the current bindcontext has been properly set.

Since contexts are completely configured at creation time, only one context table access is needed at run time for both eager and lazy binding. Thus, also evolutive software components

---

[3] Actually, CSM also allows static is-a hierarchies to be declared. Thus, when a unit hierarchy is defined statically, the corresponding context is built at compile time, so that the only run time cost is related to context retrieving from CSM KB.

can be implemented as efficiently as conservative ones (see [5] for a comparison). This overcomes one of the main problems of previous contextual programming implementations, without excluding further optimizations for eager calls, as discussed in [6]. On the other hand, this technique is flexible enough to be uniformly applied to both static and dynamic unit composition. [2] describes a highly optimized approach to inheritance between logic theories, tailored to the static case: however, it implies a rather strong modification of the host system.

On the contrary, with respect to the "minimal-intrusion" implementation requirement, CSM computation evolves as a normal Prolog computation until a particular CSM operator or an undefined predicate exception is caught. In this case, a non-local call is performed using the current (bind)context as binding environment; then, when the call has been bound and the correspondingcontext switch been done, computation proceeds as a normal SICStus Prolog one, till the next CSM event.

Thus, CSM "overrides" SICStus Prolog only in the case of non-local calls, and no overhead is introduced by local CSM computations with respect to SICStus native mechanisms. Although a reasonable degree of efficiency is achieved, the cost of a non-local call is not negligible. Global efficiency in the execution of a CSM program can be appreciably affected by the number of non-local calls performed by the program. At any rate, since this number is strongly related to program design and architectural issues (such as granularity of unit definitions), giving quantitative figures is not straightforward. Moreover, the extra-cost of non-local binding seems unavoidable, at least when dynamic inheritance is concerned.

4 Conclusions

Thanks to efficient, reliable, compiled implementations of Prolog, logic languages can be effectively exploited to develop applications according to a declarative style, so as to reduce the gap between software specification and implementation. CSM contextual extensions aim at enhancing Prolog with concepts and mechanisms for programming-in-the-large, giving effective support to evolutive criteria of software development. In fact, CSM contexts can work as structured objects, to be first used in a prototyping phase, and later refined, in order to match requirement adjustment during application development. Software evolution can then take place incrementally with no gap (and no need for extra-language tools) between the definition of prototypes and final implementation.

As far as CSM implementation is concerned, traditional techniques for extending Prolog turn out to be either quite inefficient or intrusive. Moreover, requiring a full integration with the host system of choice involves proper reusing and reengineering of existing components. Our choice to explore the so-called "program-representation based" implementation approach within the SICStus Prolog system, allowed us to meet important requirements, such as efficiency, non-intrusiveness of the extension, and full integration with SICStus programming tools. The resulting CSM run-time support delays, until context creation time, the allocation and configuration of those descriptors usually built up at compile-time by more (class-based) static languages. In this way most of the object-oriented notions (such as object creation, message passing, inheritance) have a close counterpart in the CSM contextual model, which extends its scope to advanced features such as hypothetical reasoning and dynamic (re)classification.

In the version discussed in this paper, CSM does not touch on the problem of objects with mutable state, which is the main issue of our current research. First results on this topic have been illustrated in [10].

Acknowledgements

This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of Italian National Research Council (CNR) under grant nº 890004269. We would like to thank Evelina Lamma, Paola Mello and Michele Bugliesi for many useful discussions we had with them, and DS Logics who sponsored part of the CSM development.

References

[1] A. Brogi, E. Lamma, P. Mello. A General Framework for Structuring Logic Programs. C.N.R. Technical Report "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", No 4/1, May 1990.

[2] M. Bugliesi, G. Nardiello. SelfLog: Language and Implementation. In these proceedings.

[3] E. Denti, E. Lamma, P. Mello, A. Natali, A. Omicini. Techniques for Implementing Contexts in Logic Programming. In E. Lamma and P. Mello, editors, Proceedings of the Third Workshop on Extensions of Logic Programming WELP92, Bologna, Italy, February 1992. LNAI, volume 660, pages 339-359 , Springer-Verlag.

[4] Rapid Prototyping. Computer, volume 22, No. 5, IEEE Computer Society, May 1989.

[5] B. Jayaraman, G. Nadathur. Implementation Techniques for Scoping Constructs in Logic Programming. In K. Furukawa, editor, Proceedings of the Eighth International Conference on Logic Programming, Paris (F), The MIT Press, Cambridge, Massachusetts, 1991.

[6] E. Lamma, P. Mello A. Natali. An Extended Warren Abstract Machine for the Execution of Structured Logic Programs. Journal of Logic Programming 1992, No. 14, pages 187-222, Elsevier, New York.

[7] P. Mello, A. Natali, C. Ruggieri. Logic Programming in a Software Engineering Perspective. In E.L. Lusk and R.A. Overbeek, editors, Proceedings of the North American Conference on Logic Programming NACLP89, The MIT Press, Cambridge (USA), 1989.

[8] L. Monteiro, A. Porto. Contextual Logic Programming. In G. Levi and M. Martelli, editors, Proceedings of the Sixth International Conference on Logic Programming, The MIT Press, Cambridge (USA), 1989.

[9] M. Meier, J. Schimpf. An architecture for Prolog Extensions. In E. Lamma and P. Mello, editors, Proceedings of the Third Workshop on Extensions of Logic Programming WELP92, Bologna, Italy, February 1992. LNAI, volume 660, pages 319-338, Springer-Verlag.

[10] A. Natali, A. Omicini. Objects with State in Contextual Logic Programming. Proceedings of the Fifth International Symposium on Programming Language Implementation, Tallin, Estonia, August 1993.

[11] Swedish Institute of Computer Science. SICStus Prolog User's Manual. Kista, Sweden, January 1993.