ASAP

IST-2001-38059

Advanced Analysis and Specialization for Pervasive Systems

Prototype Simulation Tool

Deliverable number:	D18
Workpackage:	Semantics-Based Modelling in CLP (WP5)
Preparation date:	1 February 2005
Due date:	1 February 2005
Classification:	Public
Lead participant:	Roskilde Univ
Partners contributed:	Univ. of BristolUniv. of Southampton, Roskilde Univ

Project funded by the European Community under the "Information Society Technologies" (IST) Programme (1998–2002).

Short description:

Aims of Work Package 5. This document is a short introduction to a software deliverable, D18: Prototype Simulation Tool. It is an outcome related to all three tasks in WP5, Analysis of Process Languages, Analysis and Specialization of Low Level Abstract Machines and Analysis of High Level Specification Languages. We aim to highlight the main features of software tools for simulation developed in the ASAP project, and give examples of some applications. As can be seen below, we have performed experiments in each of the areas represented by the three tasks, namely a process language (Petri nets), a low-level language (the PIC processor) and a specification language (B).

The aims of this deliverable are not to demonstrate the analysis results, but to show how the tools allow such languages to be simulated in CLP, allowing for further analysis by CLP tools.

WP5: Semantics-based Modelling in CLP has as its main aim the application of analysis and specialization tools for CLP to a range of other formalisms. The general method of achieving this consists of a number of stages. Let L be a target language. We speak of L-programs, though in general L need not be a programming language, but could also be a specification language, knowledge representation language, a formal logic, and so on.

1. Define abstract syntax for L and provide a systematic translation from concrete L-programs or expressions into the abstract syntax. The abstract syntax consists of tree terms that can then be straightforwardly mapped to CLP terms.

2. Provide semantics for L. A variety of semantic styles can be used: small-step (structural) or large-step (natural) operational semantics, or denotational semantics, for example. Less formal approaches to semantics, such as writing an interpreter and validating its behaviour could also be used, but the ideal is to start from formal semantics.

3. Code the semantics as a CLP program. Emphasis should be on semantic clarity rather than efficiency. Denote the CLP program capturing L's semantics as M_L.

4. Specialize M_L with respect to a particular object program in L, say P. For the purposes of simulation, the specialized program should satisfy a requirement: it should bear a structural resemblance to P, so that program points in P can be identified systematically with program points in the specialized program. In order to achieve this, the specialization process should completely specialize away the parsing of the abstract syntax [5].

5. The resulting specialized program can then be either run, in order to simulate concrete execution of P, or analysed, model-checked or further specialized.

The general rationale for this approach is that *general-purpose* analysis and specialization tools can be applied. If new versions of the tools had to be developed for each language L, there would little advantage in using CLP. Our aim is to put all the effort of developing tools into the CLP tools. Applying them to different target languages via simulators ideally means that the only extra effort to analyse a different target language is to map its syntax into CLP data structures and to express its semantics as a CLP program. Both of these tasks can be done fairly systematically, though it is a non-trivial effort for "real" languages, in our experience.

CLP support for simulating concrete computations. Using the specialized program to simulate concrete computations presents its own challenges, and again CLP provides appropriate assistance. Semantics-based interpreters are often highly non-deterministic and sometimes would not give results when run in a "naive" way. In Prolog, using a left-to-right depth-first strategy could lead to looping or unfair preference for certain computation paths. Techniques such as constraint-based execution and tabling can allow a wider range of language definitions to be simulated.

Core Tools. In general, the analysis and specialization tools consist of the full range of analysis and specialization tools developed in ASAP. We highlight the following tools are especially relevant for obtaining the specialized program meeting the requirements stated above.

- The LOGEN offline partial evaluator [4]: Offline specialization is suited for specializing semantic interpreters, since the interpreters tend to have a regular structure and it is relatively easy (though still not trivial) to find the right annotations.
- The Binding Time Analysis tool associated with Logen [1]: The BTA tool can be used either automatically, to generate annotations completely automatically, or else it can be used just to propagate and check consistency of annotations provided by the user.
- Analysis tools based on regular types [3]. LOGEN provides a straightforward generalisation mechanism driven by *filters*. These are efficient but can lose information that is vital to detect control flow in certain interpreters. The use of regular approximation performed after specialization by LOGEN has proved to be a useful technique for regaining this information and detecting unreachable parts of the flow-graph. A typical case where this technique is required occurs in languages with procedure calls, where the return address is pushed onto a stack. Upon encountering a return command, the stack has to be examined.

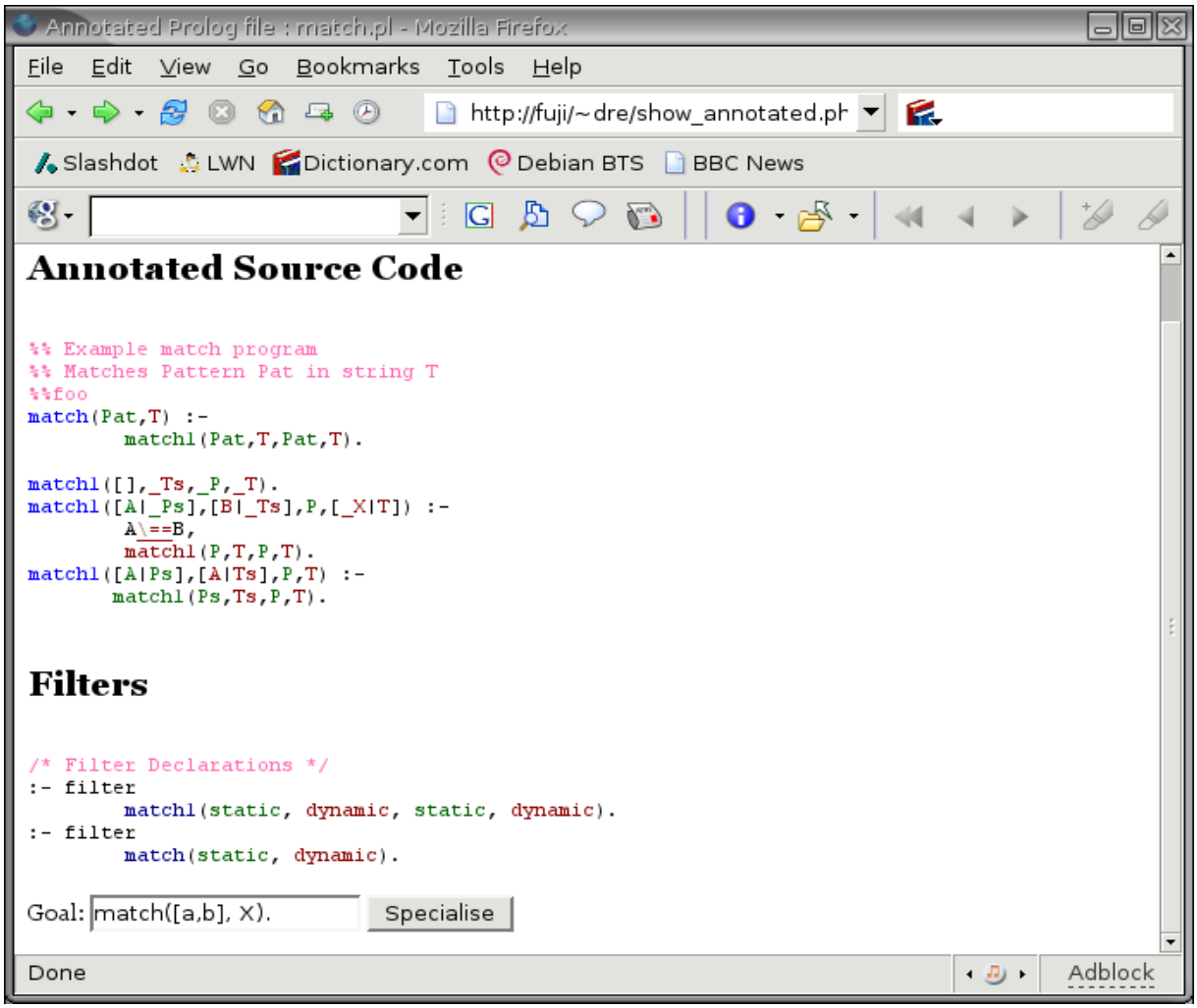

Figure 1: Displaying Annotations of a Program to Be Specialized

However, the stack is unbounded in the presence of recursive calls: over-abstraction of the stack results in loss of the return address.

Web interface to LOGEN A web interface to LOGEN is under development. This will allow the user to send their own files to be specialized via a web browser, examine the binding type and control annotations, and then run the specializer. The residual program is then produced and displayed. Two screen shots are shown in Figure 1 and Figure 2.

Experiments. We outline a number of experiments that we have carried out within the ASAP project.

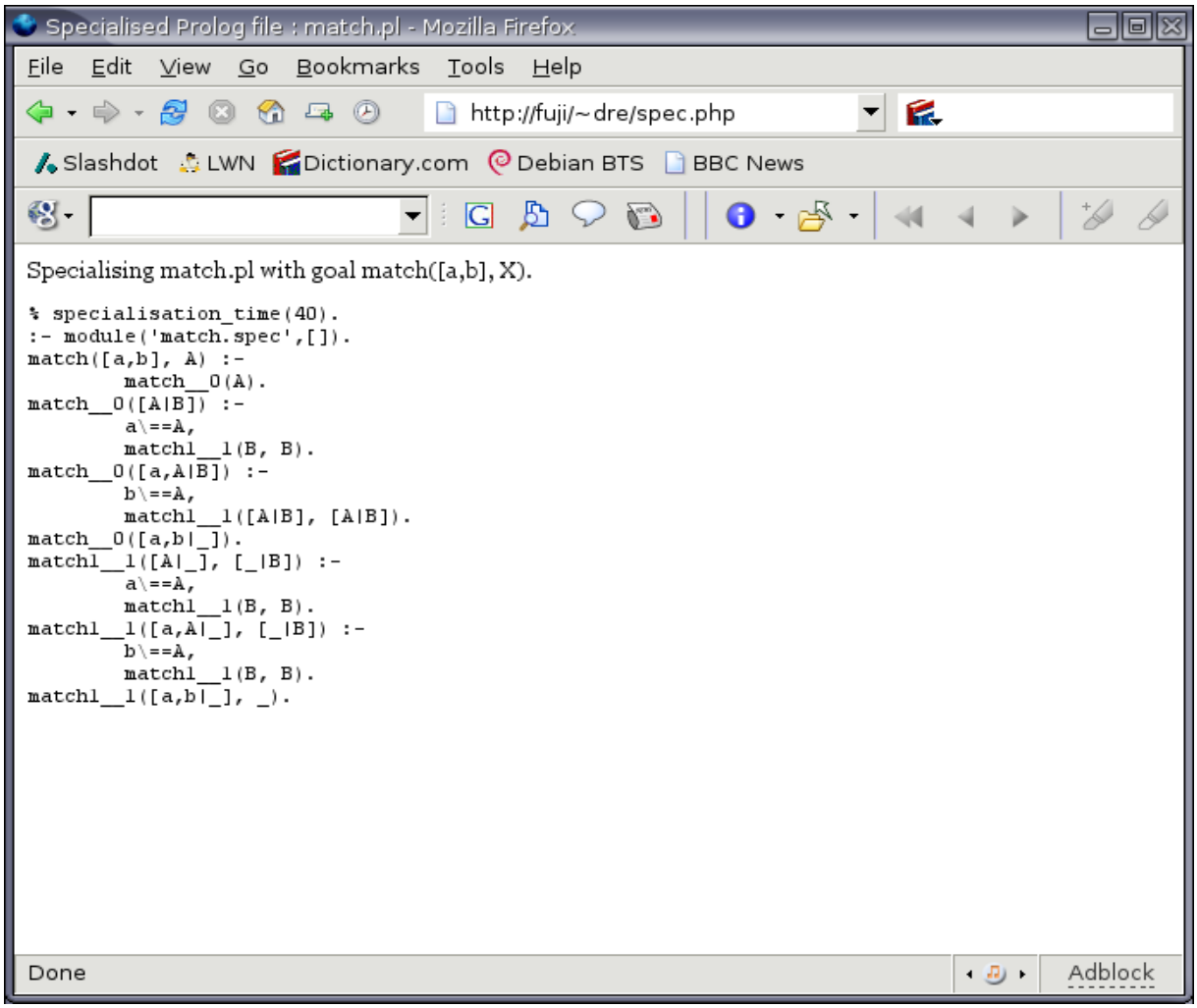

Figure 2: Displaying Specialized Program

PROB The aim of this work is to simulate B machines in CLP, and perform model-checking using CLP analysis and specialization tools. In previous work we have presented the PROB animator and model checker. Based on Prolog, the PROB tool supports automated consistency checking of B machines via *model checking*. For exhaustive model checking, the given sets must be restricted to small finite sets, and integer variables must be restricted to small numeric ranges. This allows the checking to traverse all the reachable states of the machine. PROB can also be used to explore the state space non-exhaustively and find potential problems. The user can set an upper bound on the number of states to be traversed or can interrupt the checking at any stage. PROB will generate and graphically display counter-examples when it discovers a violation of the invariant. PROB can also be used as an animator of a B specification. So, the model checking facilities are still useful for infinite state machines, not as a verification tool, but as a sophisticated debugging and testing tool.

OPNs Object Petri nets (OPNs) provide a natural and modular method for modelling many real-world systems. We give a structure-preserving translation of OPNs to Prolog by encoding the OPN semantics, avoiding the need for an unfolding to a flat Petri net. The translation provides support for reference and value semantics, and even allows different objects to be treated as copyable or non-copyable. The method is developed for OPNs with arbitrary nesting. We then apply logic programming tools to animate, compile and model check OPNs. In particular, we use the partial evaluation system LOGEN to produce an OPN compiler, and we use the model checker XTL to verify CTL formulae. We also use LOGEN to produce special purpose model checkers. We present two case studies, along with experimental results. A comparison of OPN translations to MAUDE specifications and model checking is given, showing that our approach is roughly twice as fast for larger systems. We also tackle infinite state model checking using the ECCE system. A full description of this work is published [2].

CPNs Coloured Petri Nets (CPNs) are widely used for specifying aspects of embedded systems. We defined an interpreter for CPNs, and succeeded in specializing it with respect to specific CPNs. An ongoing case-study is a CPN model of a Stack-Based Ceiling Priority protocol in a real time kernel for embedded systems, called Hartexμ, and verify certain key properties. we used *Logen* to specialize the CPN interpreter for this application.

PIC The functionality of a classic PIC processor, commonly used in applications such as wearable computing, has been modeled as an emulator written in Prolog. The PIC emulator can be specialised using an online or offline partial evaluator, which are part of an analysis and

specialisation toolset for logic programs developed in the ASAP project. The PIC emulator is described as one of the case studies in the ASAP project (deliverable D14).

The program is specialised with respect to a given program. Analysis techniques can be applied to the specialised emulator in an attempt to discover properties of the PIC program, such as constant or undefined register values, timing and synchronization when connecting more than one PIC processor running concurrently and communicating - and detection of dead code and other forms of redundancy.

Summary The ASAP tools have been demonstrated as being capable of generating CLP programs semantically equivalent to programs in a variety of languages, both high- and low-level. Indeed, gaining experience with different languages has been one of the most useful results of these tasks.

The aims of this work package, using a general purpose meta-language to implement other languages, via partial evaluation of interpreters, have been put forward in one form or another for at least 20 years. We do not claim to have solved the problems in this field. However, note that we are not tackling the full problem of automatic generation of language implementations from semantic definitions. This goal remains some way off. The immediate challenge is to show that we have tools to produce workable CLP representations of a variety of sophisticated languages, and perform non-trivial analyses on the specialized programs.

Attachments:

The software delivered consists of the LOGEN specialization tool, and its associated Binding Time Analysis system, and the regular approximation tool. These tools may be downloaded from the ASAP Project web site.

References

[1] S.J. Craig, John P. Gallagher, M. Leuschel, and Kim S. Henriksen. Fully automatic binding time analysis for Prolog. In Sandro Etalle, editor, *Pre-Proceedings, 14th International Workshop on Logic-Based Program Synthesis and Transformation, LOPSTR 2004, Verona, August 2004*, pages 61–70, 2004.

[2] Berndt Farwer and Michael Leuschel. Model checking Object Petri Nets in Prolog. In Eugenio Moggi and David Scott Warren, editors, *PPDP*, pages 20–31. ACM, 2004.

[3] J. P. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages (PADL'02)*, LNCS, January 2002.

[4] M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using the hand-written compiler generator LOGEN. *Elec. Notes Theor. Comp. Sci.*, 30(2), 1999.

[5] Michael Leuschel, Stephen-John Craig, Maurice Bruynooghe, and Wim Vanhoof. Specialising interpreters using offline partial deduction. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 340–375. Springer, 2004.