Animating Formal Models in a Communicating Sequential Process Platform

Ilias Sakellariou[1], George Eleftherakis[2], Ioannis Vlahavas[1], and Petros Kefalas[2]

[1] Department of Informatics, Aristotle University of Thessaloniki,
54124 Thessaloniki Greece
{iliass,vlahavas}@csd.auth.gr
[2] Computer Science Department, City Liberal Studies,
Affiliated College of the University of Sheffield,
13 Tsimiski Str., 54624, Thessaloniki, Greece.
{eleftherakis,kefalas}@city.academic.gr

Abstract. The X-machine formal method forms the basis for a specification/modeling language with a substantial potential value to software engineers. An X-machine is a more expressive and flexible state machine, capable of modeling both the dynamic and the static aspect of a system. Communicating X-machines provide a methodology for building communicating systems out of existing stand-alone X-machines. However, for practically using the model in an real-world system development process, a tool for demonstrating and informally verifying the properties of the modeled system is required. An ideal platform for efficiently implementing such a tool, should support, process oriented programming, efficient communication primitives and declarativeness. CSPCONS is a distributed CLP platform that supports program execution over multiple independent sequential CLP processes that synchronize though message and event passing. The present paper demonstrates the applicability of the CSPCONS programming model to the implementation of a communicating X-machine animator tool that will act as the basis for an extended set of tools that will support the formal mathematical analysis of the specified X-machine models.

Keywords: Formal Methods, Animator Tool, Logic Programming

1 Introduction

The extensive use of computers in all aspects of every day life and industry and the fact that the required control functions increasingly demand more complex software, necessitate the need for research toward the improvement of the computerised systems development process. The application of formal methods to the development process of critical systems appears to be a promising solution [1].

One of the main problems in such a development process is ensuring correctness of system specifications. This issue is of vital importance, since errors in the specification have a significant impact to the project's success. Due to the

size and complexity of systems under development, the requirements elicitation process must be supported by a set of tools that will automatically animate the formal model. The benefit is twofold. From the one side, developers can informally verify that the model simulates the actual system under development. From the other side, they can demonstrate the model to the users, aiding them to identify any misconceptions regarding the user requirements.

X-Machines is a formal method possessing the computation power of Turing machines and, since they are more abstract, they are expressive enough to be closer to the implementation of a system. This feature makes them particularly useful for modeling and also facilitates the implementation of various tools, making the development methodology built around X-machines more practical. Communicating X-machines is a set of X-machines that interact through explicit message passing. In communicating X-machines the designer can separately specify the components and how these components communicate, thus allowing a disciplined development of large and complex systems and also reuse of X-machine models, since only the specification of their communication component needs to be altered.

An ideal tool for animating communicating X-machines must necessarily support asynchronous execution of each X-machine and an accurate simulation of the message passing mechanisms. Additionally, such a tool should be able to animate X-machines by reading the corresponding models from a standard description language. Although the development of such a tool is possible in any of the languages available today, its implementation would be significantly facilitated if the underlying platform supports process based programming, advanced communication primitives and the necessary facilities to easily encode parsing of the specification language for the animation of the actual model. In this paper we argue that an appropriate platform for building such a tool is CSPCONS [2].

CSPCONS is a constraint logic programming language that follows the standard PROLOG syntax and supports process oriented programming. Applications in CSPCONS are actually collections of independent sequential PROLOG processes that interact through messages and events. The language offers a plethora of features such as synchronization through message passing, real time event driven processes, communication over TCP/IP networks, etc. The rich program and communication primitives together with the unparalleled suitability of logic programming for constructing (any) language parser make the language an excellent platform for building X-machine animator tools. This paper aims to present how the notions of X-machines nicely fit to the programing model of CSPCONS, by presenting an implementation of the X-machine animator tool in CSPCONS.

The rest of this paper is organized as follows. Section 2 reports other approaches to animator tools for formal methods. Section 3 is a brief introduction to the X-machine formal method and its extension communicating X-machines. Section 4 briefly presents the features of the CSPCONS platform, necessary for understanding the animator implementation. Section 5 describes in detail how the current X-machine animator tool was modeled in CSPCONS. Finally, section 6 concludes the paper and describes future directions of our current research.

2 Animating Formal Models

There is a plethora of tools that aim at increasing the productivity and accuracy in all the phases of the formal development of systems, through model checking or formal verification. These tools support a wide variety of methods such as OBJ[3], VDM [4], Z [5], X-machines [6], etc.

Animating formal specifications has been identified as a valuable tool since it allows for an initial evaluation of the system's specifications, detects problems through interactive model testing and requires no extensive expertise on the method applied from the user's side [7]. Of course the speed and ease of use comes at the cost that animation is by no means a complete method: its ability to detect errors depends on the set of tests that the user performs. Still, interest in the area is significant in the recent years with a number of tools reported in the literature. For instance, Possum [8] is an animator for the SUM specification language. PiZA [9] is an animator for Z specifications based on a Z to Prolog translation scheme. Pipedream [7] explores Z specifications by translating them to a first order theory and then uses the logic/functional programing language Mercury for animation. The B-Model animator [10] aims at animating model based specifications following the B-method. The IFAD VDM++ tools [4] have an animator (interpreter) that can be used to test specifications. ProBE [11] is an animator tool for CSP processes, that allows to explore the events that lead from a process state to another.

Finally, an initial version of an X-machine animator tool is presented in [6], along with a number of other tools. However, this work involved the implementation of a sequential non process based version, where message passing and X-machine execution relied on explicit execution cycles that imposed a synchronization that could potentially lead to problems or even wrong conclusions when modeling large complex systems. The work described in this paper, extends the existing set of tools by presenting an implementation of an X-machine animator tool in Cspcons, in order to simulate more accurately the asynchronous behavior of the components (X-machines) of the specified system.

3 X-machines

An X-machine is a general computational machine introduced by Eilenberg [12] and extended by Holcombe [13]. X-machines employ a diagrammatic approach of modeling the control by extending the expressive power of the FSM and model non-trivial data structures as a typed memory tuple. Therefore, X-machines are capable of modeling both the data and the control by integrating methods, which describe each of these aspects in the most appropriate way.

X-machines apply to similar cases as Statecharts and other similar notations, such as SDL, do. However, X-machines have other significant advantages. Firstly, they provide a mathematical formalism for modeling a system and a model checking method for X-machines is devised [14] that facilitates the verification of safety properties of a model. Secondly, they offer a strategy to test the implementation against the model [13], which is a generalization of W-method for FSM

testing, proved to guarantee correctness if certain assumptions in the implementation hold [15]. Thus, X-machines not only provide a modeling formalism for a system but can be used as a core notation around which an integrated formal methodology of developing correct systems is built. In principle, X-machines are considered a generalization of models written in similar formalisms, since concepts devised and findings proved for X-machines form a solid theoretical framework that can be adapted to other, more tool-oriented methods, such as Statecharts or SDL.

Extremely useful in practice is the class of so called stream X-machines, defined by the restrictions on the underlying data set, involving input symbols, memory values and output symbols. In this paper with the term X-machine we refer to the stream X-machine version. The formal definition of a deterministic stream X-machine [13] is an 8-tuple, $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ where:

- Σ, Γ is the input and output finite alphabet respectively,
- Q is the finite set of states,
- M is the (possibly) infinite set called memory,
- Φ is the type of the machine $\mathcal{M}$, a finite set of partial functions ϕ that map an input and a memory state to an output and a new memory state, $\phi : \Sigma \times M \rightarrow \Gamma \times M$
- F is the next state partial function that given a state and a function from the type Φ, denotes the next state. F is often described as a transition state diagram. $F : Q \times \Phi \rightarrow Q$
- q_0 and m_0 are the initial state and memory respectively.

Starting from the initial state q_0 with the initial memory m_0, an input symbol $\sigma \in \Sigma$ triggers a function $\phi \in \Phi$ which in turn causes a transition to a new state $q \in Q$ and a new memory state $m \in M$. The sequence of transitions caused by the stream of input symbols is called a computation. The computation halts when all input symbols are consumed. The result of a computation is the sequence of outputs produced by the sequence of transitions.

A Communicating X-machine model consists of several X-machines, which are able to exchange messages. The approach used in this paper is analytically presented in [16] and preserves to a great extend the standard theory described earlier. This version of communicating X-machines views the communicating system as a result of a sequence of operations that gradually transform a set of X-machines to a system model, leading towards a methodology of developing large-scale communicating systems. Since the communicating X-machine model is viewed as the composition of X-machine type (X-machine type is defined as an X-machine without an initial state and initial memory) with the initial memory and an initial state as well as with a set of input/output streams and associations of these streams to functions, the development of a model can be mapped into three distinct actions: (a) develop X-machine type models independently of the target system, or use existing models as they are, as components, (b) create X-machine instances of those types and (c) determine the way in which the independent instance models communicate, forming the X-machine component

($\mathcal{XMC}$). Finally, the communicating X-machine $\mathcal{CXM}$ is defined as a tuple of n $\mathcal{XMC}$ as $\mathcal{CXM} = (\mathcal{XMC}_1, \mathcal{XMC}_2, ..., \mathcal{XMC}_n)$.

In this approach X-machines have their own standard input stream but when they are used as components of a large-scale system more streams may be added whenever it is necessary. The number of streams associated with one $\mathcal{XMC}$ depends on the number of other $\mathcal{XMC}$s, from which it receives messages (Fig. 1).

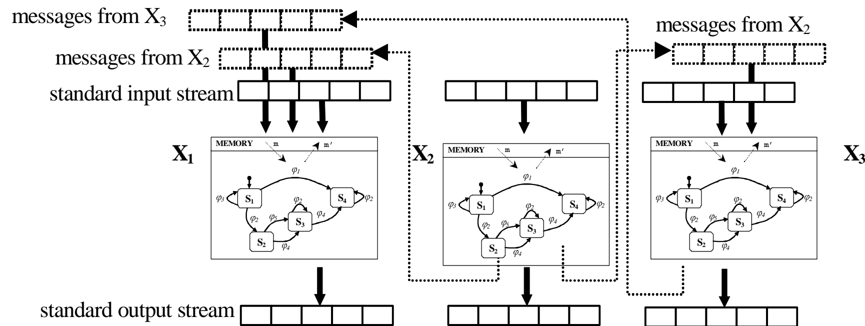

Fig. 1. Three communicating X-machine components $\mathcal{XMC}_1$, $\mathcal{XMC}_2$, and $\mathcal{XMC}_3$ and the resulting communicating system where $\mathcal{XMC}_2$ communicates with $\mathcal{XMC}_1$ and $\mathcal{XMC}_3$, while $\mathcal{XMC}_3$ communicates with $\mathcal{XMC}_1$.

In addition, for a formal method to be practical, it needs certain tools that will facilitate modeling. A prerequisite for those tools is to use a standard notation to describe models in that formal language, other than any ad-hoc mathematical notation. The formal definitions of the X-machines can be presented using the notation of X-machine Description Language which is intended to be an ASCII-based interchange language between X-machine tools [6]. Briefly, XMDL is a non-positional notation based on tags, used for the declaration of X-machine parts, e.g. types, set of states, memory, input and output symbols, functions etc.

XMDL has been enriched with syntax that provide the ability to define instances of X-machine types. Finally, XMDL provides syntax to express the solid circle and the solid diamond that are attached to the functions of the communicating machine and denote input and output streams respectively.

4 Cspcons: A Communicating Sequential LP Platform

Cspcons [2] is a programming language, targeted towards distributed (constraint) logic programming applications. It is an extension of the Communicating Sequential Prolog II (Csp-ii) language [17] and inherits all its advanced communication features and at the same time supports constraint programming. The latter follows the tradition of classic CLP languages [18].

The main feature of both Csp-ii and Cspcons systems is support for the communicating sequential process [19] programming methodology in a Prolog environment. Thus, a Cspcons application consists of a set of user defined, independent Prolog processes, each with its own execution environment, dynamic database and constraint store, that run in parallel. Processes can influence each other only by the available communication methods i.e. message passing and event generation. Fair execution on a single processor host is ensured by a time-sharing scheduler.

Processes are defined as the execution flow of a Prolog goal and can be self-driven (normal) processes or event-driven (real time). A *self driven* process is characterized by its (Prolog) goal; upon its creation the process initiates the execution of this goal. Such a process "lives" in the system until its goal terminates. Thus, self driven processes can be considered as simple Prolog threads executing a goal and are the simplest kind of the two type of processes offered. That is why they are usually referred as *normal* processes.

On the other hand, *real time* processes are more complicated and their execution is considerably different than that of a simple Prolog thread. The main feature of real time processes is that their execution is driven by events that are generated in the application, i.e. after their initialization they switch to a cyclic behavior, controlled by incoming events. For every real time process, the incoming events are gathered in a separate first-in-first-out input queue, local to each process, from which they are consumed by initiation of their private event handling (Prolog) goal.

Inter-process communication is achieved by *synchronous message passing* or by *event generation*. Messages can be only passed through one way *communication channels* and can be any Prolog term except a single unbound variable. Channels act as system-wide available resources, identified by unique names and may appear and disappear dynamically during the program's lifetime.

Events serve for triggering real time processes and are also identified by system-wide unique names. They can be generated explicitly by built-in predicates (`generate_event/2`) or implicitly by the internal clock of the scheduler. The latter allows to invoke execution of the real-time process in specific time intervals. Every event occurrence can have an optional data argument (an arbitrary Prolog term) that can be used to provide additional information.

The process-based model of Cspcons in conjunction with the declarative style of both the logic and constraint programming paradigms supported, allows the elegant implementation of any parallel or distributed application.

5 Animating X-Machines in Cspcons

The ideal animator tool should support the asynchronous execution of the X-machines, simulate efficiently their communication model and of course animate models defined in the XMDL specification language. Thus there are three main requirements for the implementation of the tool:

– Ensuring asynchronous execution.

– Preservation of the semantics of the X-machine communication.
– Automatic generation of the code for animation from XMDL specifications.

We argue that the process based programming and communication facilities of Cspcons provide an excellent platform for building such a tool. X-machines participating in the model can be implemented as independent Cspcons processes, ensuring asynchronous execution. Communication between them can be efficiently implemented using the facilities offered by the language. Finally, through the use of Prolog DCG grammar tools, the XMDL language can be efficiently parsed to create the corresponding runnable code.

Thus, in the current implementation each X-machine is modeled as an independent real-time Cspcons process, called henceforth X-process. The separation of dynamic databases of Prolog processes greatly facilitates the implementation of the animator tool, since state information and the memory status of each X-machine are stored locally in each X-process. Message passing between X-processes is based on event generation, as is described in section 5.1.

Finally, a compiler automatically translates the XMDL specification to Cspcons code, ready for execution by the animator. The translator uses DCG grammars for efficiently parsing the XMDL specification and is heavily based on the XMDL to Prolog compiler reported in [6]. The overall system architecture is depicted in figure 2.

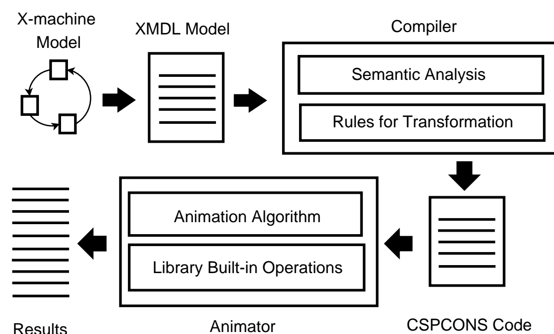

Fig. 2. X-machine Animator Architecture

5.1 X-Process Communication

The *receive* operation in X-machines is synchronous, i.e. each X-process should perform a blocking receive until a message arrives in the queue. This blocking receive operation is also *selective*: successful reception of the message does not only depend on the message format and the message queue on which it arrived, but also on a guard test performed by the corresponding function of the X-machine

in the current state. Otherwise the message should be retained in the incoming queue, since it might be consumed at a later computation state of the X-machine. The *send* operation on the other hand is asynchronous: X-machine communicate a message and continue their execution without waiting for the corresponding receive operation to occur. Although, such a communication model can be implemented in CSPCONS by message passing over channels, as demonstrated in similar cases [20], the resulting implementation is somewhat complex, since it would require the definition of two CSPCONS processes for each X-Machine.

Thus, in the X-Machine animator tool, message communication relies on an event generation scheme. Message passing involves the sender X-process generating an appropriate event that is handled by the receiver X-process (Fig. 3). Each such event, named *com-event* henceforth, carries a data argument that represents the message broadcast, annotated by the name of the sender X-process. Since com-events can be generated anywhere in the CSPCONS application and are only handled by the specified real time process, the sender only needs to be aware of the name of the receiver X-process' com-event. To simplify the communication scheme even more and avoid any necessary directory facilities etc., the name of the com-event is identical to the name of the X-process, thus each sender only needs to know the name of the receiver to broadcast the message.

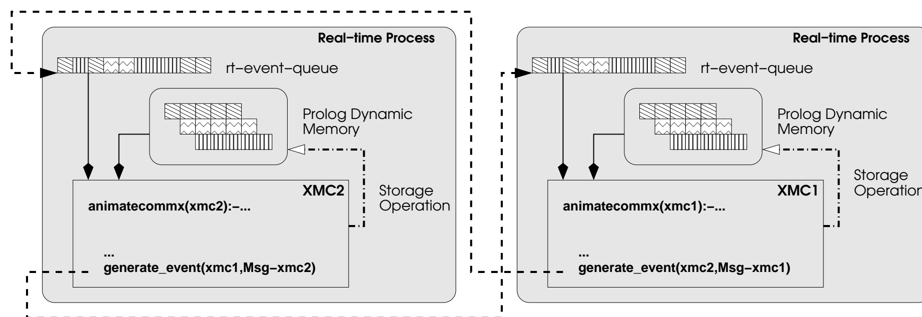

Fig. 3. X-machine Processes and Communication

Annotating each message with the name of the sender process allows the simulation of multiple input messages streams. All incoming events carrying messages are stored in a first-in-first-out manner in the real-time process queue and are consumed by the event handling goal of the X-process (Fig 3). Failure to consume a message on the specific computation state of the X-process is followed by a *storage operation* of the message to the dynamic memory of the process, from which they can be later consumed. This schema results to no message loss as well as preservation of the order that the messages arrive in the X-machine queues.

5.2 X-machine Processes

Simple communication was not the only motivation behind modeling X-machines as real-time CSPCONS processes. The cyclic behaviour of this type of process maps naturally to the execution model of each communicating X-machine, since the latter remains idle, i.e. in the same computation state, as long as an appropriate message (i.e. com-event) arrives.

Thus, X-processes are driven by their com-events. Initially, each X-process initializes its memory and computation state according to the specification and resume a cyclic behaviour to handle the incoming com-events. Upon the reception of a com-event the X-process checks the applicability of any functions in its current computation state. If there are no applicable functions, then the com-event is removed by the process queue and stored in the dynamic memory of the CSPCONS process (Fig. 3). If there are applicable functions in the current computation state, then one of these functions is executed and a transition of the machine to a state occurs. After the successful completion of the transition the X-process checks the applicability of past unconsumed messages in the new computation state and applies any functions to trigger any other transitions.

The PROLOG code that implements this cyclic behaviour is presented in Fig. 4. The build-in predicates `get_event/2` collect the next event in the event queue but remove it from the queue only upon successful completion of the `animatecommx/1` predicate. The `next_x_state/3` predicate checks the applicability of a function in the current computation state given the new message that arrived and succeeds if a transition occurs in the current state, or fails otherwise. Finally, the `checked_arrived_message/1` predicate performs any transitions triggered by previously arrived messages.

```
%%% Applicable function check and state transition.
%%% Upon a transition, the predicate checks for past messages.
animatecommx(Model):-
   get_event(Model,Input-Stream),
   current_computation_state(Model,State,Memory),
   next_x_state(Model,[State,Memory],Input-Stream),
   check_arrived_messages(Model).

%%% A not consumed message is stored in Prolog memory.
animatecommx(Model):-
   get_event(Model,Input-Stream),
   assertz(input_stream_unconsumed(Input-Stream)).
```

Fig. 4. CSPCONS PROLOG Code Implementing the X-Process

The environment is also modeled as a real-time process, generating appropriate com-events for the X-processes participating in the model under investigation. Currently, environment com-events can be either given by the user or

generated by PROLOG code. For the latter however the user has to implement the code manually. We are currently working on the definition of a modeling language for environment behaviour specification scenarios and generation the corresponding code automatically.

6 Conclusions

The communicating X-machines can be used as a core formal notation for the description of complex large systems, around which an integrated formal methodology of developing correct systems is built, ranging from model checking to testing [14]. The animator implemented provides an initial tool that facilitates informal verification of the proposed model of the system under development for user desired properties and enhances the communication between the users and the development team. It also forms the initial basis on which several tools could be built providing more functionality like formal verification (e.g. model checking) of system properties (e.g. safety, liveness, deadlock freedom), production of a complete test set etc.

The preliminary implementation of X-Machines in CSPCONS, shows that the programming facilities of the platform map very well to the requirements of the animator tool. The notion of independent PROLOG processes allows asynchronous execution of the X-machine components of the model under investigation and the communication primitives of the language allow to efficiently implement message passing between these. Additionally DCG grammars allow to arrive in a simple manner from XMDL specifications to executable code.

As stated one of the first extensions in the current tool involves a specification language to describe environment scenarios that will allow the user to automate completely the animation process. Towards this direction real time features of CSPCONS provide an excellent tool to build time based environment interaction with the communicating X-machine model.

Furthermore, we are currently investigating a new communicating X-machines model in which memory locations will be constraint domain variables with relations underlying the possible values assigned. Given that CSPCONS supports constraint programming, the tools for this new model will be directly implemented as an extension of the current tool.

References

1. Falla, M.: Advances in Safety Critical Systems: Results and Achievements from the DTI/EPSRC R& D Programme in Safety Critical Systems. (1997)
2. Vlahavas, I., Sakellariou, I., Futo, I., Pasztor, Z., Szeredi, J.: CSPCONS: A Communicating Sequential Prolog with Constraints. In: Methods and Applications of Artificial Intelligence, Procs of the 2nd Hellenic Conference on AI, SETN 2002. Volume 2308 of LNCS., Springer (2002) 72–84
3. Diaconescu, R., Futatsugi, K.: CafeOBJ Report, The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. AMAST Series in Computing 6, World Scientific. World Scientific (1998)

4. Elmstrøm, R., Larsen, P.G., Lassen, P.B.: The IFAD VDM-SL toolbox: a practical approach to formal specifications. SIGPLAN Not. **29** (1994) 77–80
5. Saaltink, M.: The Z/EVES System. In: ZUM '97: Proc. of the 10th International Conference of Z Users on The Z Formal Specification Notation, London, UK, Springer-Verlag (1997) 72–85
6. Kefalas, P., Eleftherakis, G., Sotiriadou, A.: Developing Tools for Formal Methods. In: 9th Panhellenic Conference on Informatics, Thessaloniki (2003) 625–639
7. Kazmierczak, E., Winikoff, M., Dart, P.: Verifying model oriented specifications through animation. In: APSEC '98: Proc. of the Fifth Asia Pacific Software Engineering Conference, Washington, DC, USA, IEEE Computer Society (1998) 254
8. Miller, T., Strooper, P.: Animation can show only the presence of errors, never their absence. In: ASWEC '01: Proc. of the 13th Australian Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2001) 76
9. Hewitt, M.A., O'Halloran, C., Sennett, C.T.: Experiences with PiZA, an Animator for Z. In: ZUM '97: Proc. of the 10th International Conference of Z Users on The Z Formal Specification Notation, London, UK, Springer-Verlag (1997) 37–51
10. Waeselynck, H., Behnia, S.: B model animation for external verification. In: ICFEM '98: Proc. of the Second IEEE International Conference on Formal Engineering Methods, Washington, DC, USA, IEEE Computer Society (1998) 36
11. Formal Systems (Europe) Ltd.: ProBE Tool. (2003) Available via http://www.fsel.com.
12. Eilenberg, S.: Automata Machines and Languages. Volume A. Academic Press (1974)
13. Holcombe, M., Ipate, F.: Correct Systems: Building a Business Process Solution. Springer Verlag, London (1998)
14. Eleftherakis, G.: Formal Verification of X-machine Models : Towards Formal Development of Computer-Based Systems. PhD thesis, Univ. of Sheffield, UK (2003)
15. Holcombe, M., Ipate, F.: An integration testing method that is proved to find all faults. International Journal of Computer Mathematics **63** (1997) 159–178
16. Kefalas, P., Eleftherakis, G., Kehris, E.: Communicating X-machines: from theory to practice. In Manolopoulos, Y., Evripidou, S., Kakas, A., eds.: Advances in Informatics. Volume 2563 of LNCS. Springer-Verlag (2003) 316–335
17. Futo, I.: A Distributed Network Prolog System. In: Proc. of the 20th International Conference on Information Technology Interfaces, ITI 99. (1998) 613–618
18. Jaffar, J., Maher, M.J., Marriott, K., Stuckey, P.J.: The semantics of constraint logic programs. Journal of Logic Programming **37** (1998) 1–46
19. Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM **21** (1978) 666–677
20. Sakellariou, I., Vlahavas, I., Futo, I., Pasztor, Z., Szeredi, J.: Communicating Sequential Processes for Distributed Constraint Satisfaction. Information Sciences (2005) to appear.