A Rule Language for Interaction

Carlos Castro¹, Eric Monfroy², and Christophe Ringeissen³

¹ Universidad Técnica Federico Santa María, Valparaíso, Chile
Carlos.Castro@inf.utfsm.cl

² IRIN, Université de Nantes, France
Eric.Monfroy@irin.univ-nantes.fr

³ LORIA – INRIA-Lorraine Villers-lès-Nancy, France
Christophe.Ringeissen@loria.fr

Abstract. In this paper, we propose a rule language to design interactive component languages and basic coordination languages. In our language, concurrent rules manage interactions of call-back functions that apply on a store of data. This store can freely be structured as an array, a list, a set of communication channels, etc. Our rule language can be seen as an abstract machine to specify and implement interactive component languages. We also propose such a component language devoted to solver cooperations and solver cooperation languages. We illustrate the use of this specific component language to implement some primitives of an existing solver cooperation language.

1 Introduction

The rule-based programming paradigm was introduced in the early seventies and more recently there has been a revival interest in this approach. As an example of these new developments we can mention the ELAN language [5] which is based on term rewriting: a program consists of a set of conditional rewrite rules which are controlled by strategies to specify their order of application. Another example is the CHR language [8] which extends the syntax of a host language (such as ECLiPSé, Sicstus, or Java) with rules for rewriting constraints.

Solver cooperation is a research topic that has been investigated during the last years by the Constraint Programming community [9,15,12]. Considering that very efficient constraint solvers are currently available, the challenge is to make them cooperate in order to 1) solve hybrid problems that cannot be treated by a single solver, 2) solve distributed problems (either naturally distributed or for some security and confidentiality reasons), 3) improve solving efficiency, and/or 4) reuse (parts of) solvers to reduce implementation costs. Solver cooperation languages (such as the ones proposed in [13,7,10]) provide one with primitives to manage cooperation. However, implementing such languages is a

⋆ The authors have been partially supported by the Franco-Chilean INRIA-CONICYT project Cocars, by the European IST project COCONUT, and by the Chilean National Science Fund through the project FONDECYT 1010121.
tedious task since instructions are complex and do not make a clear separation between computation, strategies, and interaction/coordination.

More recently, a component-based framework has been proposed for designing and implementing constraint solver cooperations and solver cooperation languages \cite{12}. Combining these basic components into patterns enables one to manage computation, control, and coordination required for solver cooperations. The interactive behavior of these components is described by concurrent rules.

In \cite{12}, the main motivation is to provide a simple and basic component language for describing solvers involved in cooperations, and for controlling their interactions. In this paper, we extend the ideas of \cite{12} and we propose a rule language for designing and implementing such interactive component languages and basic coordination languages \cite{14}.

In our language, rules manage interactions of call-back functions that apply to a store of data. This store can freely be structured as an array, a list, a set of communication channels, etc. Our rules apply concurrently, and informally their behavior is the following: if it can be checked that some data (possibly schematized by variables) are present at some given addresses of the store, then apply some call-back functions to these data and store the results at some given addresses in the store. For example, consider that addresses of the store are ports connecting channels, and consider the following rule:

\[
\text{chk}(i, X) \Rightarrow \text{put}(o_1, f(X)), \text{put}(o_2, g(X))
\]

Then, as soon as (and as long as) some data is present on port \(i\), the rule fires: the data is stored in the local variable \(X\); the data is removed from port \(i\); functions \(f\) and \(g\) are applied to \(X\) and the result is stored (i.e., sent when using channels) on some output ports (\(f(X)\) is stored on port \(o_1\) and \(g(X)\) on port \(o_2\)). Note that \(f\) and \(g\) can be constraint solvers.

These rules can then be combined into components to provide component languages for interaction. These components have a more complex behavior than a single rule and can be seen as some wrappers providing coordination and interactive features to functions. Consider again that addresses are ports, and consider the following component made of two rules:

\[
\begin{align*}
\text{chk}(i_1, X), \text{chk}(i_2, \text{true}) & \Rightarrow \text{put}(o_1, f(X)), \\
\text{chk}(i_1, X), \text{chk}(i_2, \text{false}) & \Rightarrow \text{put}(o_2, g(X))
\end{align*}
\]

As soon as some data is present on port \(i_1\), and that the data \textit{true} or \textit{false} is present on \(i_2\), one of the two rules fires. If \textit{true} is present on \(i_2\), then \(f\) is applied to the data on \(i_1\) (this data is then removed), and the result is sent on \(o_1\). If \textit{false} is present on \(i_2\), the process is similar, but this time \(g(X)\) is sent on \(o_2\). If there is no data on \(i_1\), or no data (or data different from \textit{true} or \textit{false}) on \(i_2\), then nothing happens. Indeed, these two rules also synchronize “messages” sent on the two ports \(i_1\) and \(i_2\). We can see the interest of such a component for solver cooperation: depending on a property (received on port \(i_2\)) checked by a solver (working somewhere else at the same time) a solver \(f\) or a solver \(g\)