Mapping Timed Concurrent Constraints into Mozart-Oz (extended abstract)

T. Sjöland, P. Brand, S. Haridi, E. Klintskog

e-mail: `sjoland@it.kth.se`

Swedish Institute of Computer Science
and
Department of Teleinformatics
KTH/The Royal Institute of Technology
Electrum 204, SE-164 40 Kista, Sweden

June 20, 2000

Abstract

The concurrent constraint-programming paradigm, *ccp* [4], provides convenient synchronization mechanisms for many distributed algorithms. *tccp* is defined in [1] as a timed extension of *ccp* usable for reactive systems. We show how *tccp* can be embedded in the distributed multi-paradigm programming system Mozart-Oz [2] with a translation using the mechanisms for discrete event simulation provided in a forthcoming release.

keywords timed concurrent constraints, discrete event simulation

1 Timed Concurrent Constraint Programming

The *concurrent constraint programming* paradigm, *ccp* [4], provides natural synchronization mechanisms for expressing concurrent algorithms. In principle different constraint based solvers can be considered to concurrently solve parts of a distributed constraint problem, but more practically the paradigm is intended for the use of logical entailment, $\theta \vdash c$ [1], as a synchronisation mechanism to achieve a "declarative" expression of control in a concurrent program.

Management of time is a natural activity in many programs, particularly *reactive programs* such as control systems etc. The timed *ccp* paradigm as defined by [1] allows common control structures in reactive programming such as time-out, delay, watchdogs etc. to be expressed. In *tccp* each computation step corresponds to a clock tick, and also *absense* of information, *non-entailment* i.e. $\theta \not\vdash c$, can be utilized. The semantics assumes bounded asynchrony and maximal parallelism.

Apart from a use in specifying, and perhaps even implementing reactive systems, *tccp* could possibly extend the modeling power of constraint programming by reducing the gap between discrete event simulation and (constraint based) planning and scheduling.

The *tccp* language is formally defined in [1]:
Assuming a given cylindric constraint system **C** the syntax of *agents* is given by the following grammar:

$$\mathsf{A} ::= \mathsf{stop} \ \mid \ \mathsf{tell(c)} \ \mid \ \Sigma_{\mathsf{i}=1}^{\mathsf{n}}\mathsf{ask(c_i)} \rightarrow \mathsf{A_i} \ \mid \ \mathsf{now\ c\ then\ A\ else\ B} \ \mid \ \mathsf{A \| B} \ \mid \exists \mathsf{x\ A} \ \mid \ \mathsf{p(x)}$$

where the c, c_i are supposed to be *finite constraints* (i.e. algebraic elements) in C. A *tccp process* P is then an object of the form $\mathsf{D.A}$, where D is a set of procedure declarations of the form $\mathsf{p(x)}$:- A and A is an agent. $\Sigma_{i=1}^{n}$ is omitted whenever $n = 1$ and $\mathsf{tell(c)} \rightarrow \mathsf{A}$ is used as a shorthand notation for $\mathsf{tell(c)}\|(\mathsf{ask(true)} \rightarrow \mathsf{A})$. The operational semantics of *tccp* is expressed by a transition system $(\mathit{Conf}, \longrightarrow)$ where each transition step is assumed to take exactly one time-unit. Configurations in *Conf* are pairs of an agent and a constraint, the *store*. $\longrightarrow \subseteq \mathit{Conf} \times \mathit{Conf}$ is the least relation satisfying the rules **R1-R10** given below.

[1] true when c follows from θ

$$\textbf{R1} \quad \langle \mathsf{tell(c)}, \theta \rangle \longrightarrow \langle \mathsf{stop}, \mathsf{c} \sqcup \theta \rangle$$

$$\textbf{R2} \quad \langle \Sigma_{\mathsf{i}=1}^{\mathsf{n}}\ \mathsf{ask(c_i)} \rightarrow \mathsf{A_i}, \theta \rangle \longrightarrow \langle \mathsf{A_j}, \theta \rangle \qquad j \in [1, n] \text{ and } \theta \vdash c_j$$

$$\textbf{R3} \quad \frac{\langle \mathsf{A}, \theta \rangle \longrightarrow \langle \mathsf{A'}, \theta' \rangle}{\langle \mathsf{now\ c\ then\ A\ else\ B}, \theta \rangle \longrightarrow \langle \mathsf{A'}, \theta' \rangle} \qquad \theta \vdash c$$

$$\textbf{R4} \quad \frac{\langle \mathsf{A}, \theta \rangle \not\longrightarrow}{\langle \mathsf{now\ c\ then\ A\ else\ B}, \theta \rangle \longrightarrow \langle \mathsf{A}, \theta \rangle} \qquad \theta \vdash c$$

$$\textbf{R5} \quad \frac{\langle \mathsf{B}, \theta \rangle \longrightarrow \langle \mathsf{B'}, \theta' \rangle}{\langle \mathsf{now\ c\ then\ A\ else\ B}, \theta \rangle \longrightarrow \langle \mathsf{B'}, \theta' \rangle} \qquad \theta \not\vdash c$$

$$\textbf{R6} \quad \frac{\langle \mathsf{B}, \theta \rangle \not\longrightarrow}{\langle \mathsf{now\ c\ then\ A\ else\ B}, \theta \rangle \longrightarrow \langle \mathsf{B}, \theta \rangle} \qquad \theta \not\vdash c$$

$$\textbf{R7} \quad \frac{\langle \mathsf{A}, \theta \rangle \longrightarrow \langle \mathsf{A'}, \theta' \rangle \quad \langle \mathsf{B}, \theta \rangle \longrightarrow \langle \mathsf{B'}, \sigma' \rangle}{\langle \mathsf{A} || \mathsf{B}, \theta \rangle \longrightarrow \langle \mathsf{A'} || \mathsf{B'}, \theta' \sqcup \sigma' \rangle}$$

$$\textbf{R8} \quad \frac{\langle \mathsf{A}, \theta \rangle \longrightarrow \langle \mathsf{A'}, \theta' \rangle \quad \langle \mathsf{B}, \theta \rangle \not\longrightarrow}{\begin{array}{c}\langle \mathsf{B} || \mathsf{A}, \theta \rangle \longrightarrow \langle \mathsf{B} || \mathsf{A'}, \theta' \rangle \\ \langle \mathsf{A} || \mathsf{B}, \theta \rangle \longrightarrow \langle \mathsf{A'} || \mathsf{B}, \theta' \rangle\end{array}}$$

$$\textbf{R9} \quad \frac{\langle \mathsf{A}, \sigma \sqcup \exists_{\mathsf{x}} \theta \rangle \longrightarrow \langle \mathsf{B}, \sigma' \rangle}{\langle \exists^{\sigma} \mathsf{xA}, \theta \rangle \longrightarrow \langle \exists^{\sigma'} \mathsf{xB}, \theta \sqcup \exists_{\mathsf{x}} \sigma' \rangle}$$

$$\textbf{R10} \quad \langle \mathsf{p(x)}, \theta \rangle \longrightarrow \langle \mathsf{A}, \theta \rangle \qquad (\mathsf{p(x)} : -\mathsf{A}) \in \mathsf{D}$$

2 Mozart-Oz

Mozart [2] is a multi-paradigm language allowing multi-threaded higher order programs to be directly executed in a distributed open system. The language contains many useful derived constructs which are translated into a core language. More detailed information is available in the documents found at the web site [3]. We assume the use of a Mozart-Oz version extended with a package for discrete event simulation, to be included in the forthcoming release.

2.1 The Mozart-Oz language

2.1.1 Syntax

A program in Mozart-Oz is expressed as a query and a set of definitions. Our translation uses the following syntactic items:

S	::=	$x = t$	(ask/tell)
	|	$S_1\ S_2$	(sequence)
	|	thread S end	(parallel thread)
	|	local x in S end	(hiding)
	|	if B then S_1 else S_2 end	(conditional)
	|	cond S_1 []...[] S_n end	(non-determinism)
	|	{P X}	(procedure call)
	|	proc {P X} S end	(procedure def)

3 Mapping of *tccp* into Mozart-Oz

We show how a mapping of *tccp* could be done by a translation into executable Mozart-Oz programs.

We define the translation to Mozart-Oz constructs directly, (using both those of the Mozart core and derived constructs), and refer to [3] for more details concerning the Mozart-Oz semantics.

We need to make sure that the translation is adequate, that is that the steps of a *tccp* execution are faithfully modelled by the Mozart-Oz interpretation. A proof of the adequacy of our translation should show that the behaviour of the generated Mozart-Oz program faithfully implements the semantics of *tccp*.

3.1 Discrete event simulation in Mozart-Oz

A *tccp* execution is expressed as a sequence of steps synchronised by the global clock mechanism {T} defined in the discrete event simulation package of Mozart-Oz.

The construction should be understood as follows: First the Mozart-Oz system executes until all threads are either suspended or are delayed (have performed a {T}). The clock is then moved one step ahead and the delayed threads are awoken. This mechanism can be used to simulate a global clock, that is a synchronised rewrite step of *tccp*.

3.2 Checking entailment without blocking

To achieve our task we need to express the checking of whether a constraint is entailed, $(\theta \vdash c)$, or not $(\theta \nvdash c)$, without using the blocking ask primitive. This can be done in Mozart-Oz using spaces. We define a boolean function for checking of immediate entailment:

```
declare
% use as {Check proc {$} Constraint end}
fun {Check P}
   S = {Space.new proc{$ _} {P} end}
in
   case {Space.askVerbose S}
   of succeeded(entailed) then true
   else false
   end
end
```

3.3 Translating *tccp* into Mozart

The different syntactic forms of *tccp* are translated by the translation function ρ taking *tccp* programs into Mozart:

We assume ρ is defined for the constraint system used.

$$
\begin{array}{rcll}
\rho(\mathsf{tell(c)}) & = & \{\mathtt{T}\}\ \rho(\mathsf{c}) & (\mathbf{R1}) \\
\rho(\Sigma_{\mathsf{i}=1}^{\mathsf{n}}\ \mathsf{ask(c_i)} \rightarrow \mathsf{A_i}) & = & \underline{\mathtt{cond}}\ \rho(\mathsf{c_1})\ \underline{\mathtt{then}}\ \{\mathtt{T}\}\ \rho(\mathsf{A_1})\,\mathtt{[]}\ldots\mathtt{[]}\,\rho(\mathsf{c_n})\ \underline{\mathtt{then}}\ \{\mathtt{T}\}\ \rho(\mathsf{A_n})\ \underline{\mathtt{end}} & (\mathbf{R2}) \\
\rho(\mathsf{now\ c\ then\ A\ else\ B}) & = & \underline{\mathtt{if}}\{\mathtt{Check}\ \underline{\mathtt{proc}}\ \{\$\}\ \rho(c)\ \underline{\mathtt{end}}\}\ \underline{\mathtt{then}}\ \rho(A)\ \underline{\mathtt{else}}\ \rho(B)\ \underline{\mathtt{end}} & (\mathbf{R3\text{-}6}) \\
\rho(\mathsf{A}||\mathsf{B}) & = & \underline{\mathtt{thread}}\ \rho(\mathsf{A})\ \underline{\mathtt{end}}\ \rho(\mathsf{B}) & (\mathbf{R7,8}) \\
\rho(\exists^{\mathsf{d}}\mathsf{xA}) & = & \underline{\mathtt{local}}\ \mathtt{Y}\ \underline{\mathtt{in}}\ \rho(\mathsf{A})[\mathsf{x}/\mathtt{Y}]\ \underline{\mathtt{end}} & (\mathbf{R9}) \\
\rho(\mathsf{p(x)}) & = & \{\mathtt{T}\}\ \{\rho(\mathsf{p})\ \rho(\mathsf{x})\} & (\mathbf{R10}) \\
\rho(\mathsf{p(x)}:-\mathsf{A}) & = & \mathsf{p} \mapsto \underline{\mathtt{proc}}\ \{\$\ \rho(\mathsf{x})\}\ \rho(\mathsf{A})\ \underline{\mathtt{end}} & \text{(proc def)} \\
\rho(\mathsf{p}) & = & M[\mathsf{p}] & \text{(procedure)} \\
\rho(\mathsf{x}) & = & \mathtt{X} & \text{(var name)}
\end{array}
$$

The translation updates (using $\mapsto$) and uses a mapping M of *tccp* procedure names into procedure definitions of Mozart.

In the transformation of procedure calls (corresponding to rule **R10** of *tccp*) we introduce a delay in order to guarantee that a procedure call always takes one time unit. This is necessary in order to allow programming of watchdogs and other pre-emption mechanisms also for processes where no suspension or delay takes place, since the maximal parallelism assumption of *tccp* could otherwise give rise to uncontrollable computations. Alternatively procedure definitions can be required to always contain a guarded choice on the top level.

4 Conclusion

We have shown how *tccp* can be naturally mapped into Mozart-Oz.

5 Acknowledgement

We wish to credit the detailed and accurate comments by Luc Onana-Alima that helped improve the quality of this technical note.

References

[1] de Boer F.S., Gabbrielli M., Meo M.C. "A Timed Concurrent Constraint Language." a.k.a. "Semantics and expressive power of a timed concurrent constraint language." In *Information and Computation*. To appear.

[2] Haridi S., Van Roy P., Brand P., Schulte C. "Programming Languages for Distributed Applications." Invited paper in *New Generation Computing*, Vol. 16, No.3, pp 223-261, 1998. Ohmsa Ltd. and Springer Verlag., Tokyo.

[3] Haridi S., Schulte C., Van Roy P., et.al. "The Mozart-Oz Programming System." `http://www.mozart-oz.org`

[4] V. Saraswat. Concurrent Constraint Programming Languages. Ph.D. Thesis 1989, published by MIT Press 1993.

[5] V. Saraswat, R. Jagadeesan, and V. Gupta. Programming in timed concurrent constraint languages.