A Process Algebra Based Framework for Promise Theory

J.A. Bergstra
I. Bethke
M. Burgess

Report PRG0701 March 2007

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

tel. +31 20 525.7591
e-mail: janb@science.uva.nl

I. Bethke

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

tel. +31 20 525.7583
e-mail: inge@science.uva.nl

M. Burgess

Faculty of Engineering
University College Oslo

St Olavs Plass
0130 Oslo
Norway

e-mail: mark.burgess@iu.hio.no

Programming Research Group Electronic Report Series

A process algebra based framework for promise theory

Jan Bergstra [a,b,1] Inge Bethke [a,2,*] Mark Burgess [c,3]

[a] *University of Amsterdam, Faculty of Science, Section Theoretical Software Engineering (former Programming Research Group)*

[b] *Utrecht University, Department of Philosophy, Applied Logic Group*

[c] *University College Oslo, Faculty of Engineering*

1 Introduction

The mechanism of an autonomous agent announcing a promise towards another agent is a powerful organizational principle in the setting of computer networks. Several approaches have been used in the past to formalize such interactions of computing devices as a representation of policies and a resolve of conflicts: Burgess and Fagernes [7] represent them as graphs, Prakken and Sergot [13,14] use temporal deontic logic, Lupu and Sloman [10] propose role theory, Glasgow et al. [9] modal logic, Bandera et al. [3] event calculus and Lafuente and Montanari [11] model checking. In this paper we use process algebra [4] for the formalization of a restricted set of aspects of promises paying attention to the sequential ordering of promises between a number of parties. As an example we specify how promises may be used in coming to an agreement regarding a simple though practical transportation problem.

In the world of process algebra we can label certain communications as promises if that makes sense intuitively. Process algebra formalisms will not provide very sharp distinctions that set apart *promise acts* from all other conceivable actions, however. Modal logics are in principle better suited for the task to capture what is specific about promises, but process algebras may be more helpful to formalize the role that promises can play in specific multi-agent

* Corresponding author. Address: Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

1 E-mail: `janb@{phil.uu.nl, science.uva.nl}`

2 E-mail: `inge@science.uva.nl`

3 E-mail: `mark@iu.hio.no`

systems. The justification of the process algebra framework for promises is therefore as follows:

(1) to provide clear and formalized cases of the use of promises in some protocols that occur within multi-agent systems,
(2) to support the design and analysis of distributed protocols that make use of promises made by autonomous agents.

The process algebra framework cannot, by nature, characterize the concept of a promise in its logical essence. That is a much harder task and requires the design of specific versions of deontic logic.

2 A data type for task bodies

The data type for task bodies is depicted in Figure 1.

$$\begin{tikzcd}
 & & \mathcal{T} \\
\mathcal{TB} \arrow[urr, "t"] \arrow[loop above, "\sim"] \arrow[loop below, "\neg"] \arrow[ddrr, bend left, "p"] \arrow[ddrr, "\#"] \arrow[ddrr, bend right, "s"'] & & {} \arrow[ll, "\gamma"'] \\
 & & \\
 & & \mathbb{B} \arrow[loop right, "\neg"] \\
 & & {} \arrow[u, bend left, "\texttt{true}"] \arrow[u, bend right, "\texttt{false}"']
\end{tikzcd}$$

Fig. 1. Data type for task bodies

$\mathcal{TB}$ is assumed to be a finite set of primitives which fall into two basic complementary categories, namely into tasks for giving or taking, or *services* and *usage*. We distinguish the atom γ—the special task of compliance. Atomic tasks are assumed services rather than uses and positive, i.e. , not negated . Two operations $\sim, \neg : \mathcal{TB} \to \mathcal{TB}$ on tasks are then considered:

(1) (*usage*) if x is a task then $\sim x$[4] is the task of making use of x as performed by another agent, and
(2) (*negation*) if x is a task then $\neg x$ is the task of not doing x.

[4] In [6,7] the use of a x is denoted by $-x$ or $U(x)$ instead of $\sim x$.

Moreover, $s, p : \mathcal{TB} \to \mathbb{B}$ specify the properties *service* and *positive*. The interaction of these operations satisfies the laws in Figure 2. Note that $\neg$ is

$\sim\sim x = x$	$s(\gamma) = \texttt{true}$	$p(\gamma) = \texttt{true}$
$\neg\neg x = x$	$s(\neg x) = s(x)$	$p(\neg x) = p(x)$
$\sim\neg x = \neg\sim x$	$s(\sim x) = s(x)$	$p(\sim x) = p(x)$

Fig. 2. Interaction of usage, negation and service

overloaded in the sense that it acts as negation on tasks and on Booleans. The actual meaning, however, will always be clear from the context.

In general, promises can be viewed as declarations to keep certain tuples of data within a given range of values. Promises are thus typed. We therefore assume a collection of types, $\mathcal{T}$, and a typing function $t : \mathcal{TB} \to \mathcal{T}$ providing types for task bodies. Given a service x, we assume that types do not differ under usage or negation, i.e.,

$$t(\sim x) = t(x) = t(\neg x).$$

Furthermore, since promises can be incompatible with each other we assume a symmetric incompatibility relation $\# : \mathcal{TB} \times \mathcal{TB} \to \mathbb{B}$. We write $x\#y$—instead of $\#(x, y) = \texttt{true}$—if x and y *cannot* both be realized at the same time by the same agent. Only tasks of similar type can exclude one another. Moreover, tasks are incompatible with their negations. For incompatible tasks x and y, however, x will be compatible with $\neg y$. Observe that $\neg(x\#x)$ is derivable from

$$x\#\neg x \qquad \frac{x\#y}{y\#x} \qquad \frac{x\#y}{t(x) = t(y)} \qquad \frac{x\#y}{\neg(x\#\neg y)}$$

Fig. 3. Laws of incompatibility

the axiom and the third rule in Figure 3 using the law of double negation shift.

3 A transition system for promises

Let A be a partially ordered set containing so-called agents. For agents $a, b \in A$, we write $a \leq b$ if a is subordinated to b. We denote a promise x between arbitrary autonomous agents a and b—while being unspecific about how and when they are made—by

$$a \xrightarrow{\pi:x} b.$$

For $x \in \mathcal{TB}$ with $s(x) = \texttt{true} = p(x)$ we distinguish the 4 kinds of promises given in Figure 4 where

$$\boxed{\begin{array}{ll} (1) & a \xrightarrow{\pi:x} b \\ (2) & a \xrightarrow{\pi:\sim x} b \\ (3) & a \xrightarrow{\pi:\neg x} b \\ (4) & a \xrightarrow{\pi:\neg\sim x} b \end{array}}$$

Fig. 4. Promised exchanges of services x between autonomous agents a and b

(1) a promises b to provide its service x,
(2) a promises b to make use of its service x,
(3) a promises b not to provide its service x, and
(4) a promises b not to make use of its service x.

We tacitly assume that promises are equal under equal tasks, i.e., that

$$x = y \Rightarrow a \xrightarrow{\pi:x} b = a \xrightarrow{\pi:y} b.$$

One can generalize this basic notation of promise exchange to a more expressive system where agents can make promises about what other agents might do—provide a service or make use of. A generalized notation of the form

$$a[c] \xrightarrow{\pi:x} b[d]$$

denotes that 'a promises b that c will do x for d'. The autonomously made promises in Figure 4 are then equivalent to their more general notations

$$a[a] \xrightarrow{\pi:x} b[b].$$

If $c \leq a$ and $a[c] \xrightarrow{\pi:x} b[d]$, then the promise by a implies an obligation for c. Autonomous agents, however, ought not to be obliged to anything.

Another interaction between autonomous promises and the more general kind of promises is given by the so-called *compliance promise* between agent c and a,

$$c \xrightarrow{\pi:\gamma} a,$$

where c promises to comply with a. We then have

$$a[c] \xrightarrow{\pi:x} b[d], c \xrightarrow{\pi:\gamma} a \Longrightarrow c \xrightarrow{\pi:x} d.$$

One can consider the even more general notation

$$a[c_1, \ldots, c_n] \xrightarrow{\pi:x} b[d_1, \ldots, d_m]$$

denoting that

(1) 'a promises b that one of $c_1, \ldots, c_n$ wil do x for some one amongst $d_1, \ldots, d_m$' if x is a service, or
(2) 'a promises b that one of $c_1, \ldots, c_n$ wil make use of x as done by one amongst $d_1, \ldots, d_m$' if x is a usage

provided x is positive. In the negative case none of $c_1, \ldots, c_n$ wil do x for any of $d_1, \ldots, d_m$' etc.

In distributed systems design it is unhelpful to use either $a[c] \xrightarrow{\pi:x} b$ or $a[c] \xrightarrow{\pi:x} b[d]$. If these occur in a design they should and usually can be translated into small protocols using *voluntary* promises only. In the sequel we will therefore focus on promises of the basic form made by autonomous agents forgetting about the general notion of promises.

We will model states as sets of basic promises that do not conflict together with transition rules that describe the development of such states.

A *transition rule* for a basic promise has one of the two forms

$$pi_{a,b}(x)\ \frac{S}{S \oplus a \xrightarrow{\pi:x} b} \qquad \textit{promise introduction}$$

or

$$pw_{a,b}(x)\ \frac{S}{S \ominus a \xrightarrow{\pi:x} b} \qquad \textit{promise withdrawal}$$

where

(1) S is a state, i.e., a set of non-conflicting basic promises,

(2) the promise introduction $pi_{a,b}(x)$ labels the transition rule with the announcement that introduces the promise $a \xrightarrow{\pi:x} b$,
(3) the promise withdrawal $pw_{a,b}(x)$ labels the transition rule with the speech act that withdraws the promise $a \xrightarrow{\pi:x} b$,
(4) $\oplus$ combines the state S with the promise $a \xrightarrow{\pi:x} b$ yielding a new state, and
(5) $\ominus$ removes the promise $a \xrightarrow{\pi:x} b$ from the state S yielding a new state.

Since states are sets of non-conflicting promises, a promise event rule is applicable only if the conclusion of the rule is a set of non-conflicting promises, i.e., if for all $a \xrightarrow{\pi:y} c \in S$, $\neg(x\#y)$. Here we assume that an autonomous agent is itself responsible for making no promises that would require performing incompatible tasks ('breaking its own promises' is Burgess' nomenclature in [6]).

This system can be generalized to generalized promises. A typical rule in this format is of the form

$$pi_{a[c]\to b[d]}(x) \ \frac{S \oplus c \xrightarrow{\pi:\gamma} a}{S \oplus c \xrightarrow{\pi:\gamma} a \oplus c \xrightarrow{\pi:x} d}$$

In addition to incompatibility we now introduce *exclusiveness* $E : \mathcal{TB} \to \mathbb{B}$ marking tasks that cannot be served to or consumed from two different agents at the same time. This will mean that $a \xrightarrow{\pi:x} b$ and $a \xrightarrow{\pi:x} c$ with $b \neq c$ can never be kept if $E(x) = \texttt{true}$—and should not both be made either. In the presence of exclusiveness, the promise event rule takes the conditional format

$$(E(x) \to \forall c \neq b \ \neg pi_{a,c}(x)) \Longrightarrow pi_{a,b}(x) \ \frac{S}{S \oplus a \xrightarrow{\pi:x} b},$$

Note that exclusiveness is not related to incompatibility: 'taking a train' and 'taking a car' are conflicting tasks; 'being driven by' b, however, excludes 'being driven by' c.

4 An example

We now consider an ACP-style process algebra with the standard operators $+, \cdot, \|$ for choice, sequential and parallel composition (cf. [2,8]), and conditional guards (cf. e.g. [1]) based on atomic actions like $pi_{a,b}(x)$ and $pw_{a,b}(x)$. In such a setting a protocol, $P_{a,b}(x)$, that describes a plausible course of actions for introducing a promise $a \xrightarrow{\pi:x} b$ can be given by

$$\begin{aligned} P_{a,b}(x) = pi_{a,b}(x) \cdot (\ & (E(\sim x) \to \forall c \neq a\ \neg pi_{b,c}(\sim x)) :\to pi_{b,a}(\sim x) \\ & + \\ & pi_{b,a}(\neg\sim x) \cdot (pw_{a,b}(x) \parallel pw_{b,a}(\neg\sim x)) \\ ) & \end{aligned}$$

Here is an example from our recent experience. The autonomous agents Jan, Jürgen, and Mark consider the task of transport by car to the Jacobs University Bremen,i.e.,

(1) $A = \{ja, ju, ma\}$, and
(2) $\mathcal{TB} = \{tbc2JUB, \neg tbc2JUB, \sim tbc2JUB, \neg\sim tbc2JUB\}$.

Since one cannot be transported in 2 different cars at the same time and by 2 different people, $\sim tbc2JUB$ is exclusive, i.e.,

$$E(\sim tbc2JUB).$$

A possible execution of $P_{ja,ma}(tbc2JUB) \parallel P_{ju,ma}(tbc2JUB)$ is given by the trace

$$\begin{array}{ll} pi_{ja,ma}(tbc2JUB) & \cdot \\ pi_{ma,ja}(\sim tbc2JUB) & \cdot \\ pi_{ju,ma}(tbc2JUB) & \cdot \\ pi_{ma,ju}(\neg\sim tbc2JUB) & \cdot \\ pw_{ju,ma}(tbc2JUB) & \cdot \\ pw_{ma,ju}(\neg\sim tbc2JUB). & \end{array}$$

On an intuitive level, the trace can be described as follows: Initially Jan promises Mark a lift to JUB which Mark accepts. Then Jürgen makes this promise too which Mark—because of the exclusiveness of this task—declines. Thereupon Jürgen withdraws his offer and Mark his declination.

This kind of example can typically be found in data centre management: renaming the agents and tasks to

(1) $A' = \{user, ISPA, ISPB\}$, and
(2) $\mathcal{TB}' = \{transport\ packets, \ldots\}$

we derive an example of choosing a supplier for e.g. packet transport, power/electricity etc. Promises are then exclusive if ISPA and ISPB are competitors, for instance.

5 Conclusion

We have provided the outline of a process algebra based framework for promise theory. Using this algebra in combination with conditional guards one can formalize—as other approaches do—how promises might be used in coming to an agreement. However, in contrast to the static approaches to promise theory mentioned in the introduction, in the here chosen framework—the algebra of communicating processes ACP— the interaction of promises and the resolution of conflicts can be modelled in a dynamic way.

This formalization is treating promises at a meta-level. There are also underlying events or processes that the promises suppress—we do not talk about how the promises are kept, or comment on their reliability; that is a different matter. Thus our description is at a *promise management level.* At that level we could say it describes an autonomous process.

Acknowledgements

Jan Bergstra and Mark Burgess acknowledge helpful discussions with Jürgen Schönwälder, School of Engineering and Science, during a working visit of one week to the Jacobs University Bremen in October 2006.

References

[1] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In [5], 273–323 (1992).

[2] J.C.M. Baeten and W.P. Weijland. *Process algebra.* Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press (1990).

[3] A.K. Bandara, E.C. Lupu, J. Moffet, and A. Russo. Using event calculus to formalise policy specification and analysis. *Proceedings of the 5th International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*, IEEE Computer Society, 229–239 (2004).

[4] J.A. Bergstra, A. Ponse, and S.A. Smolka (eds). *The Handbook of Process Algebra*, Elsevier (2001).

[5] M. Broy (ed). *Programming and Mathematical Method, Marktoberdorf Summer School 1990*, NATO ASI series F 88, Springer Verlag (1992).

[6] Mark Burgess. A promise theory approach to collaborative power reduction in a pervasive computing environment. In [12], 615–624 (2006).

[7] Mark Burgess and Siri Fagernes. Promise theory—a model of autonomous objects for pervasive computing and swarms. *ICNS*, 118 (2006).

[8] Wan Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. An EATCS Series. Springer (2000)

[9] J. Glasgow, G. MacEwan, and P. Panagaden. A logic for reasoning about security. *ACM Transactions on Computer Science* 10, 226–264 (1992).

[10] E. Lupu and M. Sloman. Conflict analysis for management policies. *Proceedings of the Vth International Symposium on Integrated Network Management IM'97*, Chapman & Hall, 1–14 (1997).

[11] A.L. Lafuente and U. Montanari. Quantitative mu-calculus and ctl defined over constraint semi-rings. *Electronic Notes on Theoretical Computing Systems QAPL*, 1–30 (2005).

[12] J. Ma, H. Jin, L.T. Yang, and J. J.-P. Tsai (eds). *Ubiquitous Intelligence and Computing: Third International Conference*, Lecture Notes in Computer Science, volume 4159, Springer, (2006).

[13] H. Prakken and M. Sergot. Contrary-to-duty obligations. *Studia Logica* 57, 91–115, (1996).

[14] H. Prakken and M. Sergot. Dyadic deontic logic and contrary-to-duty obligations. In [15], 223–262 (1997).

[15] D.N. Nute (ed). *Defeasible Deontic Logic*, Synthese Library, Kluwer (1997).

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.