P2P Web of Trust Calculi and Promise Games

ND Jebessa

March 2014

Abstract

We describe the idea behind the `TelosVM` framework, a model of "computation as social agency" based on promise theory. Policy is described as a *promise program* $\mathscr{P}$. The system $\mathscr{U}$ is a network of voluntarily cooperating agents (human, machine or inanimate), each trying to keep promises that apply to its context. Peers in scope observe if an agent has kept a promise and update the web of trust accordingly. The semantics of $\mathscr{P}$ and the dynamics of $\mathscr{U}$ call for logical and probabilistic trust calculi.

1 Introduction

Several approaches as to how to manage the behavior of a network of agents have been proposed. A recent one is *promise theory* [1] which is a model of *what could happen* when autonomous agents voluntarily cooperate toward a systemic purpose[1] by exchanging promises. It bridges the worlds of intention and behavior in a principled manner that can be used with humans and technology alike. It has found application in the popular `CFEngine` *infrastructure as code* tool. Leveraging so-called *convergent operators*, the tool takes a given system from an initial state to a desired state specified as policy code.

Promise theoretic systems embrace uncertainty and do away with determinism. An agent might promise something, or another agent might make it make a promise, as in a human programming a machine. Either way, we are not sure if a promise will be kept. This might have to do with what it takes for a promise to be kept (e.g. dependence on yet unverified promises), the trustworthiness of the agent, the ability of the agent, or the effect of the environment. The symbiosis between trust and promises cannot be overestimated.

We then ask *how trustworthy a system is (or could be)*, given a behavioral description (i.e. the promise program $\mathscr{P}$, § 2.2), plus information on the system's promise universe $\mathscr{U}$ (§ 2.5) (e.g. a web of trust, agent context, minimal history or promise keeping probability distributions). We believe that the structure and interpretation of $\mathscr{P}$, together with a model of the agents in the system, will allow us to develop meaningful trust calculi. Furthermore, the same techniques employed to analyze $\mathscr{P}$ and $\mathscr{U}$ can be used to build tools for promise theory and its CFEngine-like incarnations. Earlier work [3] has shown that a definition of local and global trust based on the concept of promises is superior to one based on actions. We extend the *typed trust* techniques developed there for planar promise graphs to complex promise networks that emerge in real world

[1] *Telos* is Greek for 'purpose'. We are looking at feedback-controlled purpose [2].

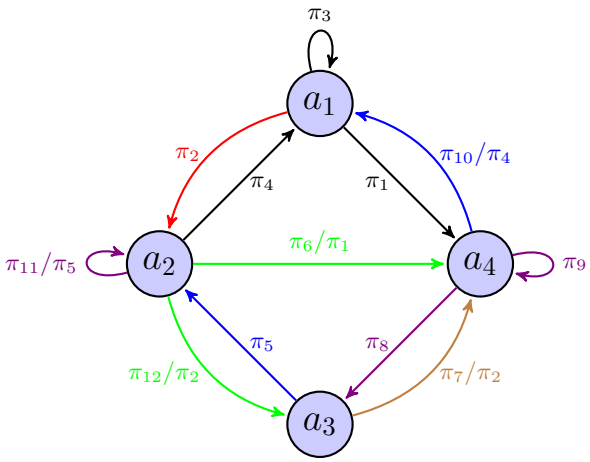


Figure 1: An abstract promise network

systems. This is particularly interesting because we can determine *who trusts who for what in what context and by how much.*

2 The TelosVM Framework

Consider the abstract promise network shown in fig. 1. It is a simplified version of a promise program with 4 abstract agents $\{a_i\}$ having made 12 promises $\{\pi_i\}$ between themselves. $\mathscr{P}$ is written in a decalarative promise language λ (§ 2.1). The body of one of the promises, π_7, is shown in listing 1.

```
1  # this is a comment
2  brown: # the promise type T(π7) is brown (see fig. 1)
3        c1&(c2|c3)&!c4|c5:: # context formula φc(π7) = c1 ∧ (c2|c3) ∧ ¬ c4|c5
4            "a3" -> "a4" #"Promiser" -> "Promisee"
5                # constraints (χ(π7))
6                # (zero or more lval => rval pairs)
7                lval1 => "rval1", # 1st constraint
8                handle => "p7", # unique promise id (π7)
9                # ...
10               depends_on => "p2", # conditional promise (i.e. π7/π2)
11               usebundle => anotherbundle, # use promise (U(π))
12               lvaln => "rvaln"; # nth constraint
```

Listing 1: Promise body for π_7 in fig. 1

The (lval, rval) pairs in listing 1 constrain the promiser agent a_3's behavior. Those constraints need to be verified by a promise keeping engine (§ 2.3.1), which also performs type checking and semantic analysis. Th engine periodically runs on a promise machine (PM) (§ 2.3) so as to keep promises that apply to its context. Peers in scope including the promiser itself voluntarily observe

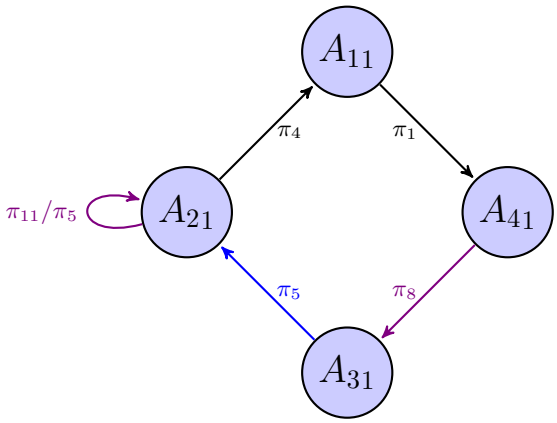


Figure 2: A promise machine PM_1 executing $\mathscr{P}\prime$ (part of $\mathscr{P}$ from fig. 1)

and report on promise state, updating the web of trust. The engine is also responsible to evaluate context formulae like φ_c above to determine if a promise is made in each promiser agent's context, and then follows a certain order to invoke for each promise the appropriate promise keeping procedure written in a possibly Turing-complete language. In a way, the promise automaton's state (§ 2.4) is the Boolean output of the procedure, i.e. 0 or 1. If not verified, the state is *unknown*.
In a system, we will have a number of PMs, each executing a part $\mathscr{P}\prime$ of a promise program $\mathscr{P}$ with its own concrete agents, as defined by a hard context specific to the PM. A promise machine executing part of $\mathscr{P}$ is shown in fig. 2. Note the *concrete agent* A_{ij} in PM j corresponding to the *abstract agent* a_i. Now imagine having a web of trust among the PMs, for example, that is based on past observations about promise keeping. Given a system $\mathscr{U}$ described as $\mathscr{P}$, we want to calculate the trustworthiness of the whole using some kind of trust calculus that operates on the parts. Furthermore, the evolution of the system from initial condition to a convergent state needs to be modeled. TelosVM provides a language, a code analysis engine (static/dynamic), and abstractions called *promise virtual machines* (§ 2.6) to look at interesting behavior of the system.

2.1 Promise Language (λ)

The language λ is inspired by CFEngine. It supports primitive promises, bundled promises, use promises, and conditional promises [1]. By using namespaces (spatial context), it implicitly supports so-called knowledge promises (i.e. knowledge of an agent about the body of a promise made by another agent).
A promise body has the form:

```
promiseType:
      contextExpression::
          "Promiser" -> "Promisee"
              # constraints (lval => rval pairs)
              # ...
```

Multiple promises could be bundled together.

```
bundle bundleType bundleName
{
    promiseType:
        Monday.!Amsterdam|Sunday.Beijing:: # context logic expression
            "Douglas" -> {"PromiseeSix", "PromiseeSeven"}
                handle => "promise42",
                lval2 => rval2,
                # more constraints here ...
                lvaln => rvaln;
    anotherPromiseType:
        (isHappy|isNotHappy.isRich).Hr07::
            "Adams" -> "Sheldon"
                lval1 => rval1,
                handle => "notFourtyTwo",
                lval3 => "rval3"
}
```

The body of promise π specifies:

1. A promise type T
2. A context formula φ_c with context variables $\{c_i\}$:
   (a) A set of hard contexts $\{c_{Hi}\}$
   (b) A set of soft contexts $\{c_{Si}\}$
3. A promiser and promisee a_S and a_R
4. A set of constraints as per the promise type, including:
   (a) `handle`: a unique promise identifier
   (b) `depends_on`: conditional promise (ordering, π_2/π_1)
   (c) `usebundle`: use promise ($U(\pi)$)
   (d) variable/context *operators*

The language offers a way to order bundles via the `bundlesequence` constraint:

```
body common control
{
    bundlesequence => {"b1", "b2", "b3"};
}
```

2.2 Promise Program ($\mathscr{P}$)

The most fundamental promise program is the *deadlock*:

```
bundle trading deadlock
{
    sell:
        moneyPaid::
            "Seller" -> "Buyer"
                handle => "aPromiseToSell",
                depends_on => "aPromiseToPay";
    pay:
        itemReceived::
            "Buyer" -> "Seller"
                handle => "aPromiseToPay",
                depends_on => "aPromiseToSell";
}
```

One of the agents should break the symmetry for the interaction to proceed. We see that trust plays a key role here.
A program $\mathscr{P}$ may have variables (*data flow*), soft contexts (*control flow* based on state), and promise automata (*system state*). Variables are linked with contexts via *operators*. Each promise may have zero or more variable-context operators. These operators are found in special type of promises (`vars` and `contexts`), and with constraints of type `and`, `or`, or arbitrary propositional logic expressions with atoms from θ (canonified variable and context values as well as promise state). Part of $\mathscr{P}$ will be executed on a promise machine.

2.3 Promise Machines

We define a promise machine PM_i as a set of concrete agents sharing a unique and valid combination of hard context variables $\{c_{Hi}\}$. Itself can be considered as a super-agent that 'physically' binds multiple agents into one promise keeping engine.
Each PM_i has an assignment θ_i of hard contexts (C_{Hi}), soft contexts (C_{Si}), and a state table for its promise automata ($\theta_{\pi i}$) (see § 2.4).

2.3.1 Promise Keeping Engine

A promise keeping engine is a collection of procedures (one per promise type) on each PM, running a subset of $\mathscr{P}$ that applies to the PM's hard context.
No matter what the initial state of the PM is, and provided that there is no conflict in constraints, the engine follows a *normal order* to converge to desired state. Other orderings are imaginable as well [4]. Following the `bundlesequence`, it tries to verify that all promises are kept.

2.4 Promise Automaton

We define a promise automaton with 3 distinct states: 0, 1, or *unknown*. The latter introduces uncertainty. This may be because the promise is not made

(i.e. φ_c is not true), or the promises it depends on or uses are yet to be verified. Every time the engine runs, it updates the state of the automaton. A minimal history of each promise automaton's evolution may be kept, correlating it with the state of other automata. $\Pr(\pi)$, the probability that a promise will be kept, depends on other promises, the trustworthiness of the promiser agent and/or the PM. In a 100% trustworthy system where there is no logical conflict between the promises (i.e., after model checking), the promise automata will all have the state 1 after convergence.

2.5 Promise Universe ($\mathscr{U}$)

A network of PMs, conversing in the language λ and having a topology σ (described using spatial hard contexts) is said to form a *promise universe*. This represents the system we are interested in reasoning about.
We could imagine arbitrary network architectures for $\mathscr{U}$: centralized, hierarchical, fully connected, P2P, et cetera. In the motivating scenario (§ 3), we consider a peer-to-peer network of agents.

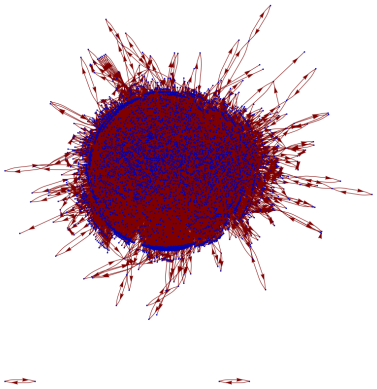
2.6 Promise Virtual Machines

Given a $\mathscr{U}$, any meaningful ensemble of promises possibly spanning multiple PMs are said to form a promise virtual machine (PVM). This is to say that the obsereved state of all promises in a PVM *together* carries with it to an observer information about a certain role or purpose of the ensemble. Of course, $\mathscr{U}$ itself is a PVM.
The concept of a PVM is important in TelosVM as it serves as a glass to look at interesting behavior of the system. For example, we might ask which agents are involved to achieve a specific task, which agents are critical (ranking), and who is trusting who for what. Because the involved promises have an underlying dependency graph, we can do further analyses [5].

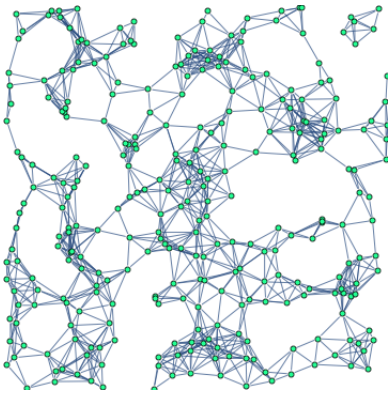
3 `The CFEther`: a Motivating Scenario

We are experimenting with a P2P game of promises that is supposed to mimic the behavior of so-called DAOs (distributed autonomous organizations). The DAO[2] infrastructure configuration is written in a stripped-down version of CFEngine. Individual promise types are assumed to be verified by decentralized applications running on top of a blockchain à la Ethereum. Ethereum, inspired by BitCoin, is described as "a platform and a programming language that makes it possible for any developer to build and publish next-generation decentralized applications." Promiser and promisees interact in a peer-to-peer manner, trading over the counter. As interactions proceed, a web of trust emerges (fig. 3a), as in BitCoin-OTC. The idea is to model a distributed human-machine system using promises (i.e. policy). By observing past behavior and employing trust calculi ([3]), we envision to calculate trustworthiness of the whole system (or parts of it, as observed through a PVM).

[2] `https://www.ethereum.org/`



(a) BitCoin-OTC Web of Trust over a 3 year period



(b) Simulated promise universe $\mathscr{U}$ with ≈300 PMs, agents sampled from fig. 3a

4 Research Approach

It goes without saying that TelosVM belongs to a class of logic-based games called *Boolean games* [6]. One major difference is that we have a programming language for game rules (i.e. λ). The promise programs specify abstract agents whose concrete counterpart in a PM have their own local propositional variables that they control [7, 6]. Model checking of simplified promise programs can be shown to be PSPACE-complete [7]. Having used modal logic to clarify how promise games in TelosVM look like, tractability issues take us to *randomized algorithms* and *probabilistic abstract interpretation.*We use a combination of static and dynamic analysis of λ code, with $\mathscr{P}$ as input and an optional $\mathscr{U}$ specification. Among the techniques we are using are model checking with abstract interpretation, Dijkstra's weakest preconditions, and MCMC sampling for probabilistic programs.

5 Related Work

Bergstra and Burgess [1] have done seminal work on promise theory, and have presented a definition of agent trust as the expectation that a promise will be kept [3]. We extend their idea further, by leveraging the semantics of $\mathscr{P}$ and dynamics of $\mathscr{U}$. In essence, we have introduced a model of computation based on promise programs with trust controlling (and emerging from) the dynamics. TelosVM can be seen as a generalization of so-called Boolean games (see [6]), which are a class of logic-based games where each agent controls a subset of propositional variables. van Benthem [8] uses "computation as social agency" to describe such unconventional models of computation.

6 Conclusion

We have described the `TelosVM` framework. The symbiosis between trust and promises, both typed, is emphasized. Trust calculi benefit from structural analysis of a promise program $\mathscr{P}$. When such a program is executed on a network of PMs, dynamics matters. The modality of promises (kept, not kept, unknown) is observed by peers in scope, thereby updating the *web of trust* on which is based our envisioned trust calculi.

References

[1] Jan A. Bergstra and Mark Burgess. A static theory of promises. *CoRR*, abs/0810.3294, 2008.

[2] A. Rosenblueth, N. Wiener, and J. Bigelow. Behavior, purpose and teleology. *Philosophy of Science*, 10:18–24, 1943.

[3] Jan A. Bergstra and Mark Burgess. Local and global trust based on the concept of promises. *CoRR*, abs/0912.4637, 2009.

[4] Steve Traugott. Why order matters: Turing equivalence in automated systems administration. In *LISA*, pages 99–120. USENIX, 2002.

[5] ND Jebessa et al. Towards purpose-driven virtual machines. In Maritta Heisel and Eda Marchetti, editors, *Proceedings of the Doctoral Symposium at the International Symposium on Engineering Secure Software and Systems (ESSoS-DS 2013)*, pages 13–19, Paris, France, Feb 2013.

[6] Ulle Endriss, Sarit Kraus, Jérôme Lang, and Michael Wooldridge. Designing incentives for boolean games. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '11, pages 79–86, Richland, SC, 2011.

[7] Wiebe van der Hoek and Michael Wooldridge. On the logic of cooperation and propositional control. *Artificial Intelligence*, 164(1–2):81 – 119, 2005.

[8] Johan van Benthem. Computation as social agency: What and how. Technical report, University of Amsterdam, 2013.