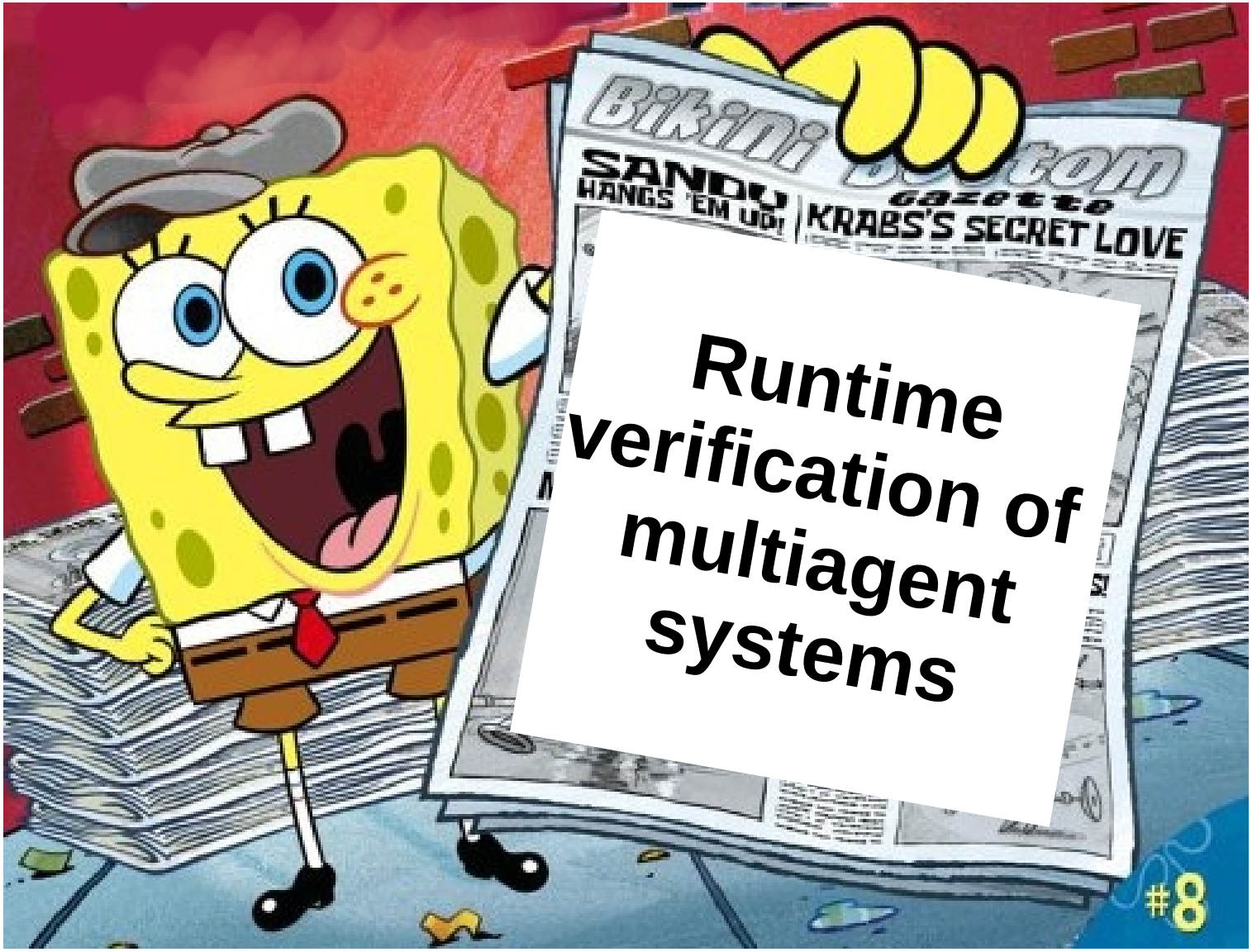


Runtime verification of multiagent systems

Davide Ancona, Daniela Briola, Angelo Ferrando, Viviana Mascardi

DRV Workshop, Bertinoro 2016

Part I

Agents and MASs & “traditional” approaches to MAS verification

Weak notion

An agent is an hardware or software system

- situated,
- autonomous,
- flexible
  - reactive
  - proactive
  - social.

N. Jennings, K. Sycara, M. Wooldridge, JAAMAS 1(1), 1998.

Strong notion

A computer system either conceptualized or implemented using concepts that are more usually applied to humans.

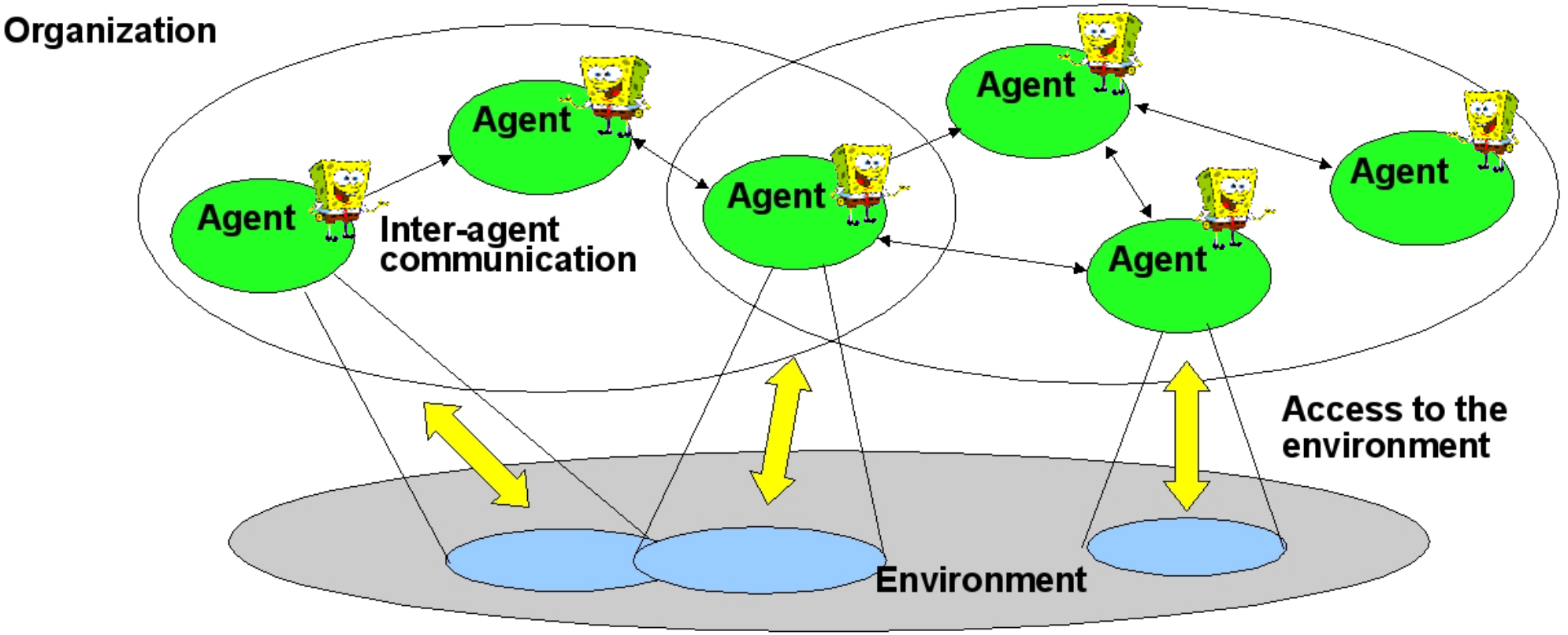
- Mentalistic notions.

Y. Shoham. Agent-oriented programming. Artificial Intelligence, 60(1):51–92, 1993.

- Emotional notions.

J. Bates. The role of emotion in believable agents. Communications of the ACM, 37(7):122–125, July 1994.

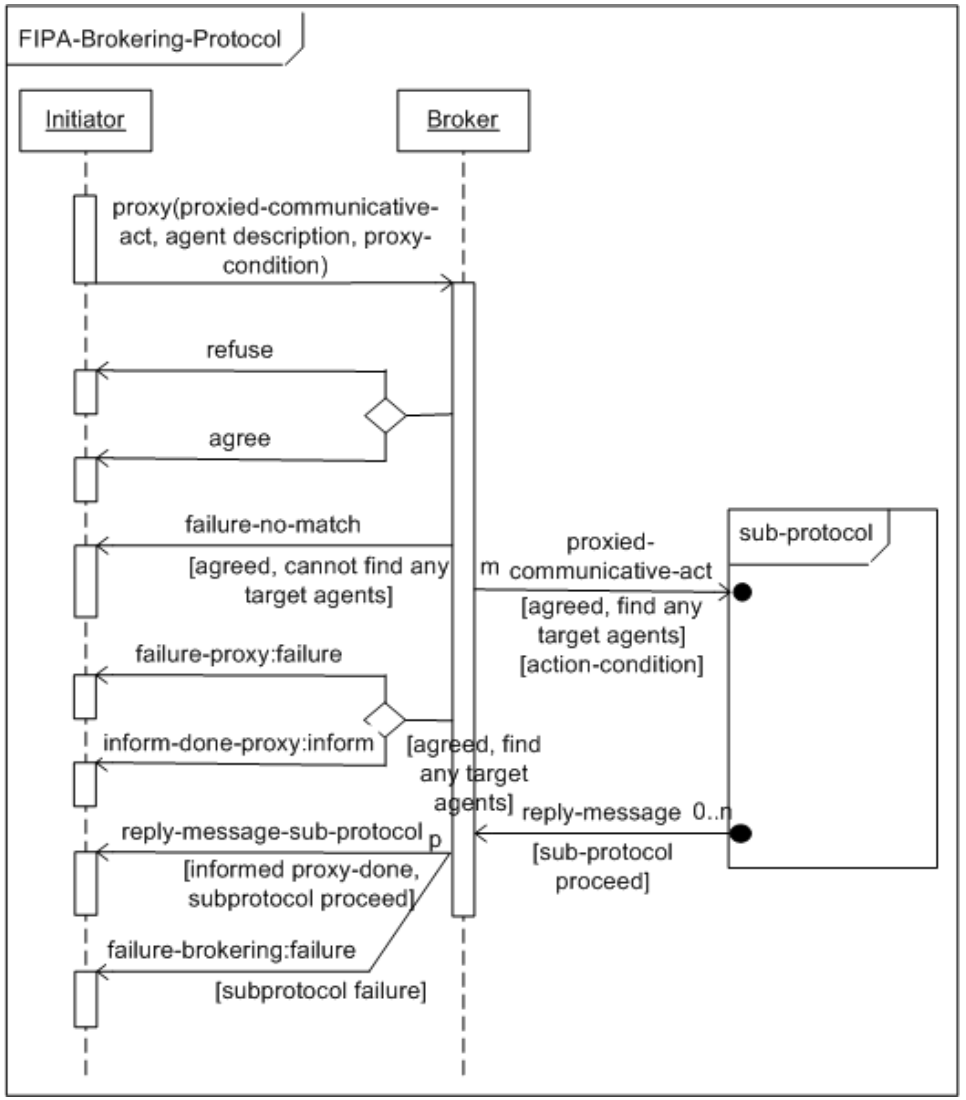
Multiagent system

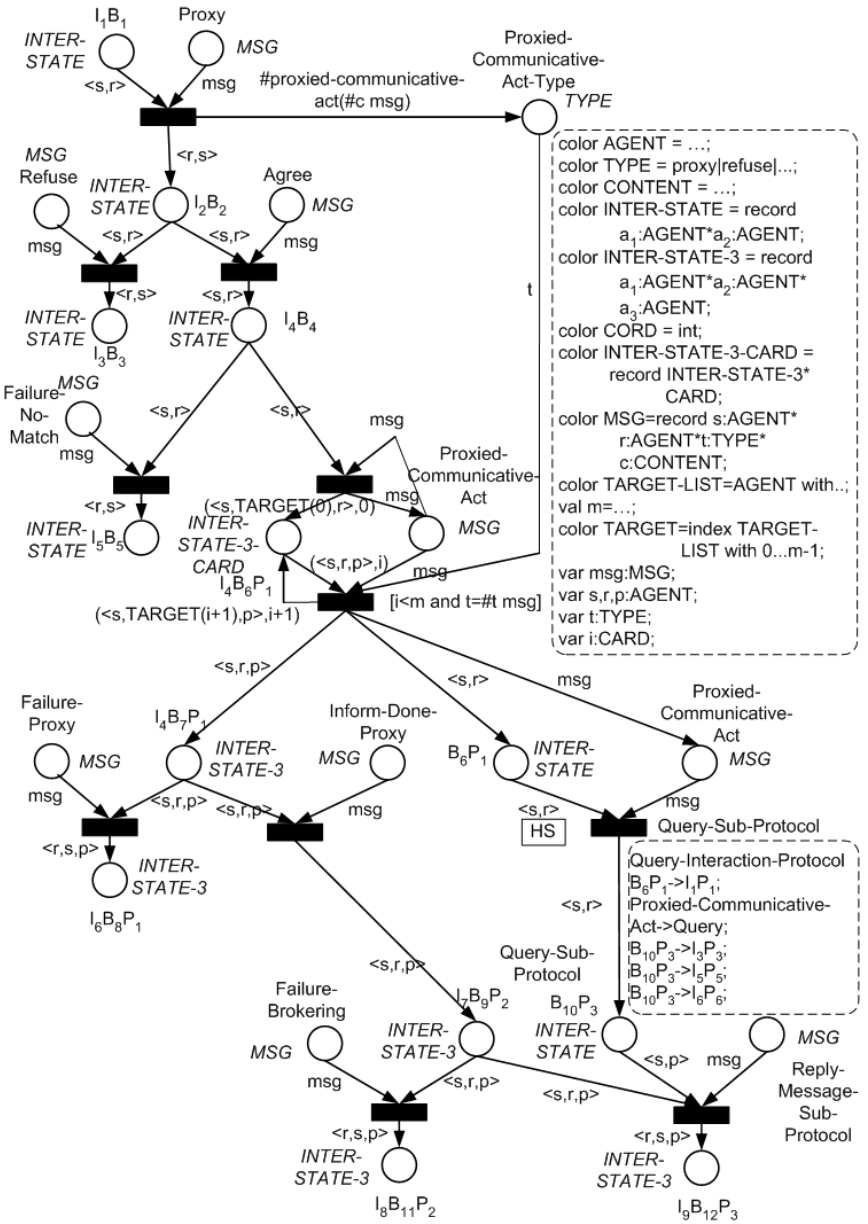


Multiagent system

- each agent has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint;
- there is no global system control;
- data is decentralized; and
- computation is asynchronous.

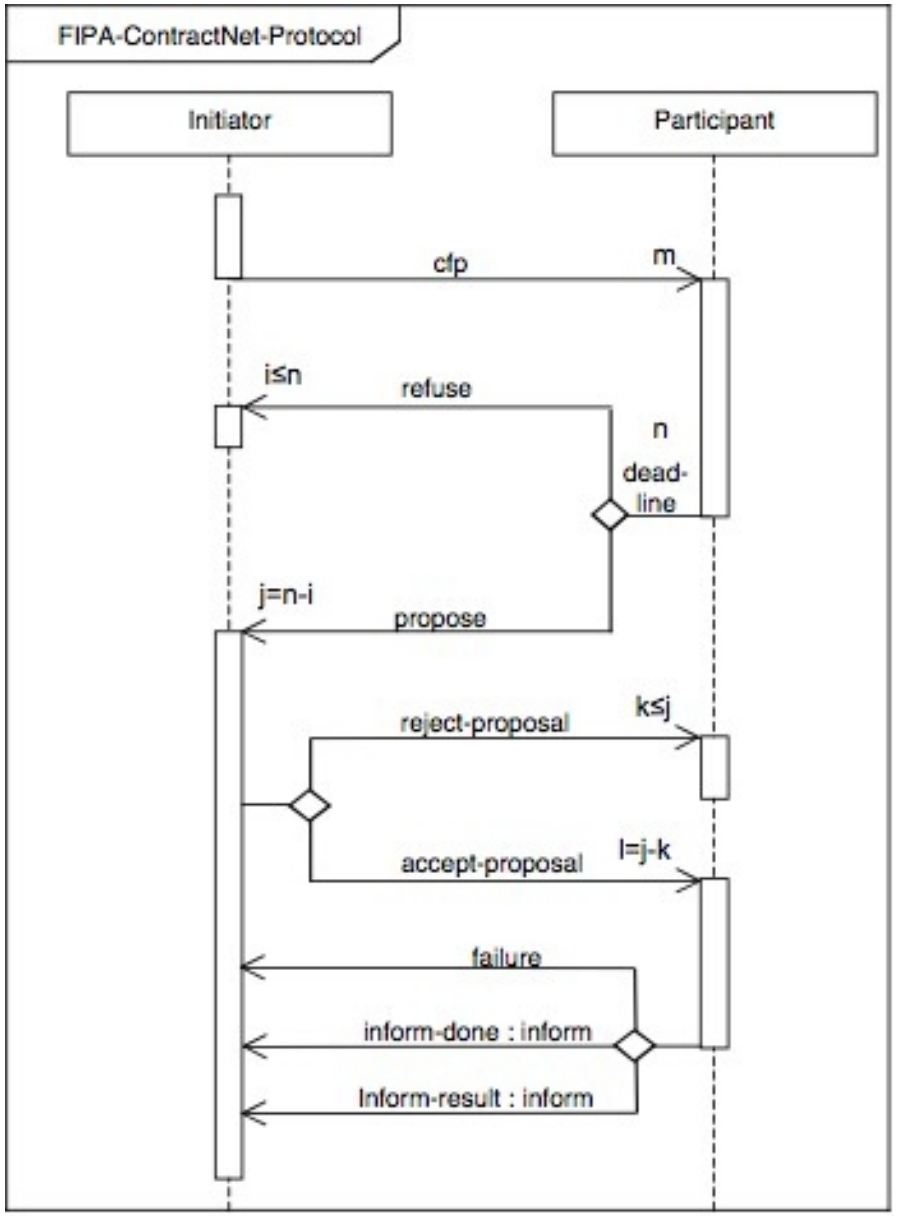
Agent interaction protocols (we go crazy for them!)





G. Gutnik, G. A. Kaminka, JAIR 2006

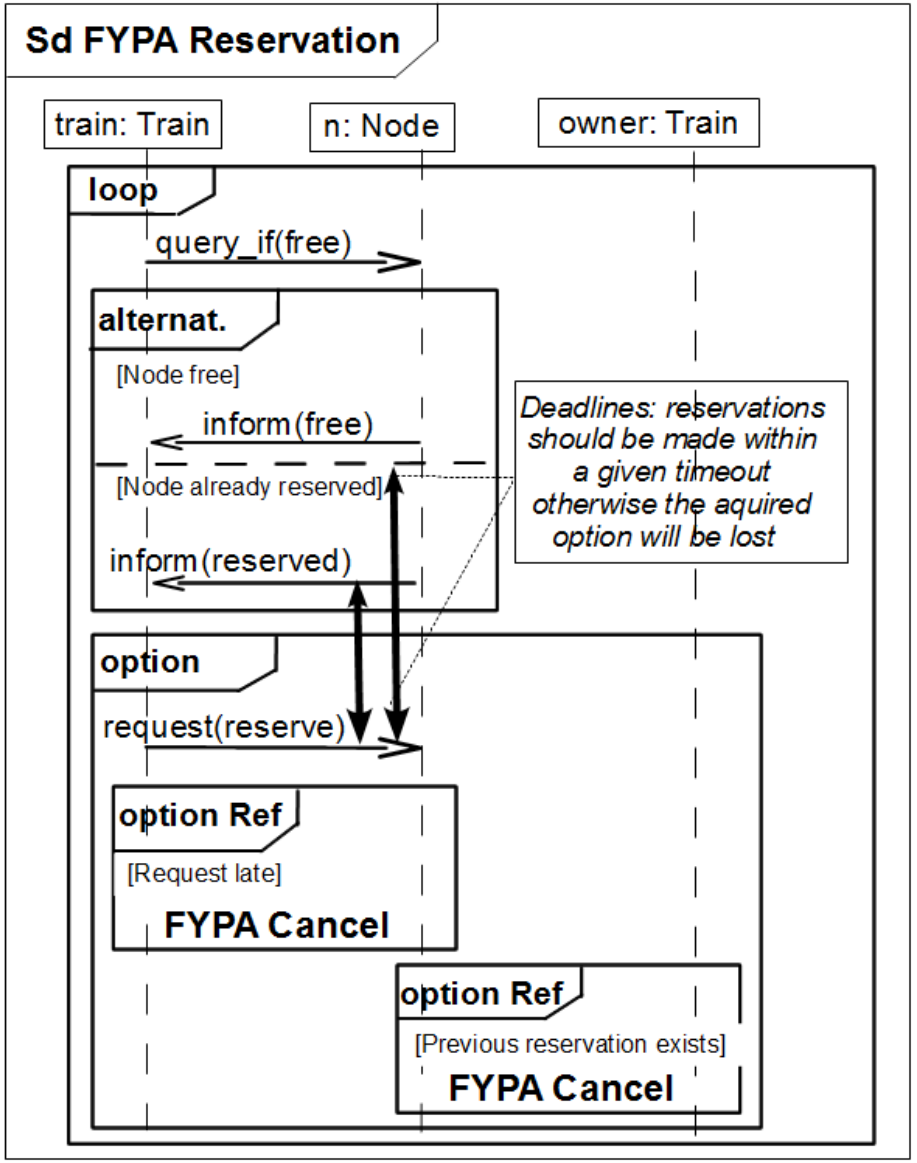
Agent interaction protocols (we go crazy for them!)




```
(omod PACKAGE-FIPA-CONTRACT-NET-PROTOCOL is
 extending TEMPLATE-FIPA-CONTRACT-NET-PROTOCOL .
 ...
 class Agent | Play-Role : Role, MBox : MB, AcqList : AcquaintanceList, State : AgentState .  *** [1]

 msg ComingMsg : Sender Receiver Act -> Msg .                          *** [2]
 vars I P : Aoid .                                          *** [3]
 ...
 crl [IsendmsgP]:  < I : Agent | PlayRole : Initiator, MBox : MB, AcqList : ACL, State : StartI >    ***[4]
          =>
            ComingMsg(I, HeadA(ACL), call-for-proposal)
           < I : Agent | PlayRole : Initiator, MBox : MB, AcqList : TailA(ACL), State : StartI >
          if ACL =/= EmptyacquaintanceList .

 rl [Preceiningmsgl]: ComingMsg(I, P, call-for-proposal)                    ***[5]
         < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : StartP >
        => Execute(P, DecisionProcess)
         < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : OfferEvaluationP > .

 crl [Pdecision1]: Execute(P, DecisionProcess) Event(P, Cond)                                  ***[6]
        < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : OfferEvaluationP >
       => ComingMsg(P, I, propose)
        < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : WaitP >
       if Cond = IsAccepted .

 crl [Pdecision2]: Execute(P, DecisionProcess) Event(P, Cond)                                  ***[7]
        < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : OfferEvaluationP >
      => ComingMsg(P, I, notunderstood)
        < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : WaitP >
       if Cond = NotClear .

 crl [Pdecision3]: Execute(P, DecisionProcess) Event(P, Cond)                                  ***[8]
        < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : OfferEvaluationP >
      => ComingMsg(P, I, refuse)
        < P : Agent | PlayRole : Participant, MBox : MB, AcqList : I, State : FailureP >
       if Cond = IsRefused .
 ...
 endom)
```

F. Mokhati, B. Sahraoui, S. Bouzaher, M. T. Kimou, JoT 2010

Agent interaction protocols (we go crazy for them!)



$ND\_QUERY = msg(Train, N, nd\_query\_if(free(infinity, T1, T2, From)), cid(CId))^1$
$D\_QUERY = msg(Train, N, d\_query\_if(free(MyPr, T1, T2, From)), cid(CId))^1$
$CONSTR\_ON\_PATH\{List, (Train, MyPr, N, T1, T2, From, CId)\} =$
$((msg(Train, N, nd\_query\_if(free(MyPr, T1, T2, From)), cid(CId)) :$
$CONSTR\_ON\_PATH\{[(Train, MyPr, N, T1, T2, From, CId)|List],_{\}} + \lambda))+$
$(msg(Train, N, d\_query\_if(free(MyPr, T1, T2, From)), cid(CId)) :$
$CONSTR\_ON\_PATH\{[(Train, MyPr, N, T1, T2, From, CId)|List],_{\}} + \lambda)))$
[the tuple (Train, MyPr, N, T1, T2, From, CId) forms a consistent path with the elements in List]
$ND\_RESERVED\{T2, NextT3\} = (msg(N, Train, inform(nd\_reserved(Ow, OwPr, Arc, From,$
$NextT3, NextT4), expires(\_TO)), cid(CId))^0 : \lambda)[NextT3 > T2]$
$D\_RESERVED\{T2, NextT3\} = (msg(N, Train, inform(d\_reserved(Ow, OwPr, Arc, From,$
$NextT3, \_NextT4), expires(\_TO)), cid(CId))^0 : \lambda)[NextT3 > T2]$
$DISASTER = msg(N, \_HumanOperator, inform(disaster(Ow, Train, T1, T2)), cid(CId))^0$
$STEAL\_ON\_TIME = (msg(Train, N, request(reserve\_on\_time(Arc, MyPr, Ow, OwPr, T1, T2),$
$expired(AbsTime)), cid(CId))^0 : REFUSE\_NODE\_ROBBED\_TRAIN)$
$STEAL\_LATE = ((msg(Train, N, request(reserve\_late(Arc, MyPr, Ow, OwPr, T1, T2),$
$expired(AbsTime)), cid(CId))^0 : (REFUSE\_NODE\_SAME\_TRAIN + \lambda)) + \lambda)$
$RESERVE\_ON\_TIME = (msg(Train, N, request(reserve\_on\_time(Arc, MyPr, none, 0, T1, T2),$
$expired(AbsTime)), cid(CId))^0 : \lambda)$
$RESERVE\_LATE = (msg(Train, N, request(reserve\_late(Arc, MyPr, none, 0, T1, T2),$
$expired(AbsTime)), cid(CId))^0 : REFUSE\_NODE\_SAME\_TRAIN)$
$REFUSE\_NODE\_SAME\_TRAIN = (msg(N, Train, refuse(reserve), cid(CId))^0 : \lambda)$
$REFUSE\_NODE\_ROBBED\_TRAIN = (msg(N, Ow, refuse(reserve), cid(\_CIdX))^0 : \lambda)$
$FREE = (msg(N, Train, inform(free(Arc, From), expires(\_TO)), cid(CId))^0 : \lambda)$
$NONDISP\_NONDISP = (ND\_RESERVED \cdot (DISASTER : \lambda))$
$WT\_DISP = (D\_RESERVED\{T2, NextT3\} \cdot ((STEAL\_ON\_TIME + STEAL\_LATE) + \lambda))$
$DISP\_NONDISP = ND\_RESERVED\{T2, NextT3\}$
$FREE\_NODE = (FREE \cdot ((RESERVE\_ON\_TIME + RESERVE\_LATE) + \lambda))$

V. Mascardi, D. Briola, D. Ancona, AI*IA 2013

Logic-based agents

Logic-based agents

“Traditional” approach to build artificial intelligent systems.

Logical formulae: symbolic representation of the agent environment and desired behavior.

Logical deduction or **theorem proving**: syntactical manipulation of this representation.

The Beliefs, Desires, Intentions (BDI) logic

Combination of:

- temporal logic (linear time in Cohen and Levesque, branching time in Rao and Georgeff)
- modal logic(s) of beliefs, desires & goals (intentions)

The modalities of Rao and Georgeff's BDI logic are BEL(φ), GOAL(φ), INTEND(φ).

P.R. Cohen and H.J. Levesque. Intention is choice with commitment. Artificial Intelligence, 1990

A. S. Rao and M. P. Georgeff. Decision Procedures for BDI Logics. Journal of Logic and Computation, 1998

The BDI architecture

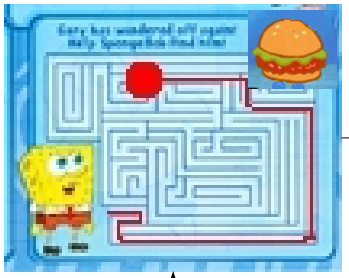
The BDI architecture is one of the best known and most studied architectures for cognitive agents.

AgentSpeak(L) is an elegant, logic-based programming language inspired by the BDI architecture.

Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Proc. of MAAMAW '96, pp. 42-55, 1996.

The BDI architecture

Plans



Intentions

Reasoner

Goals

Beliefs

Environment

Many works on static verification, both of programs and of interaction protocols...

Static verification of BDI MASs

Model Checking AgentSpeak

Rafael H. Bordini Michael Fisher Carmen Pardavila Michael Wooldridge
Department of Computer Science, University of Liverpool,
Liverpool L69 7ZF, U.K.
{R.Bordini, M.Fisher, C.Pardavila, M.J.Wooldridge}@csc.liv.ac.uk

2003

ABSTRACT

This paper introduces AgentSpeak(F), a variation of the BDI logic programming language AgentSpeak(L) intended to permit the model-theoretic verification of multi-agent systems. After briefly introducing AgentSpeak(F) and discussing its relationship to AgentSpeak(L), we show how AgentSpeak(F) programs can be transformed into Promela, the model specification language for the Spin model-checking system. We also describe how specifications written in a simplified form of BDI logic can be transformed into Spin-format linear temporal logic formulæ. With our approach, it is thus possible to automatically verify whether or not multi-agent systems implemented in AgentSpeak(F) satisfy specifications expressed as BDI logic formulæ. We illustrate our approach with a

systems start to be applied to safety-critical applications such as autonomous spacecraft control [12, 7].

Currently, the most successful approach to the verification of computer systems against formally expressed requirements is that of *model checking* [4]. Model checking is a technique that was originally developed for verifying that finite state concurrent systems implement specifications expressed in temporal logic. Although model checking techniques have been most widely applied to the verification of hardware systems, they have increasingly been used in the verification of software systems and protocols [9].

Our aim in this paper is to present model checking techniques for verifying systems implemented in AgentSpeak(L). The AgentSpeak(L) BDI logic programming language was created by Rao [13], and was later developed into a more practical pro-

Static verification of BDI MASs

Verifying multi-agent programs by model checking

Rafael H. Bordini · Michael Fisher · Willem Visser · Michael Wooldridge

2006

Published online: 24 February 2006

Abstract This paper gives an overview of our recent work on an approach to verifying multi-agent programs. We automatically translate multi-agent systems programmed in the logic-based agent-oriented programming language AgentSpeak into either Promela or Java, and then use the associated Spin and JPF model checkers to verify the resulting systems. We also describe the simplified BDI logical language that is used to write the properties we want the systems to satisfy. The approach is illustrated by means of a simple case study.

Static verification of BDI MASs

Model Checking Agent Programming Languages*

Louise A. Dennis[α] Michael Fisher[α] Matthew P. Webster[β] Rafael H. Bordini[γ]

α : Department of Computer Science, University of Liverpool, Liverpool L69 3BX, United Kingdom.
l.a.dennis@liverpool.ac.uk, mfisher@liverpool.ac.uk

β : Virtual Engineering Centre, Daresbury Laboratory, Warrington WA4 4AD, United Kingdom.
matt@liverpool.ac.uk

γ : Institute of Informatics, Federal University of Rio Grande do Sul,
PO Box 15064, 91501-970 Porto Alegre, RS - Brazil.
R.Bordini@inf.ufrgs.br

2012

Published as *Automated Software Engineering Journal 19(1):3-63*, Mar. 2012

Abstract

In this paper we describe a verification system for multi-agent programs. This is the first *comprehensive* approach to the verification of programs developed using programming languages based on the BDI (belief-desire-intention) model of agency. In particular, we have developed a specific layer of abstraction, sitting between the underlying verification system and the agent programming language, that maps the semantics of agent programs into the relevant model-checking framework. Crucially, this abstraction layer is both flexible and extensible; not only can a variety of different agent programming languages be implemented and verified, but even *heterogeneous* multi-agent programs can be captured semantically. In addition to describing this layer, and the semantic mapping inherent within it, we describe how the

Static verification of Agent Interaction Protocols

Logic-based Agent Communication Protocols

Ulle Endriss[1], Nicolas Maudet[2], Fariba Sadri[1], and Francesca Toni[1,3]

[1] Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2AZ (UK)
Email: {ue,fs,ft}@doc.ic.ac.uk
[2] LAMSADE, Université Paris 9 Dauphine
75775 Paris Cedex 16 (France)
Email: maudet@lamsade.dauphine.fr
[3] Dipartimento di Informatica, Università di Pisa
Via F. Buonarroti 2, 56127 Pisa (Italy)
Email: toni@di.unipi.it

2003

Abstract. An agent communication protocol specifies the rules of interaction governing a dialogue between agents in a multiagent system. In non-cooperative interactions (such as negotiation dialogues) occurring in open societies, the problem of checking an agent's conformance to such a protocol is a central issue. We identify different levels of conformance (weak, exhaustive, and robust conformance) and explore, for a specific class of logic-based agents and an appropriate class of protocols, how to check an agent's conformance to a protocol *a priori*, purely on the basis of the agent's specification.

Static verification of Agent Interaction Protocols

Choice, Interoperability, and Conformance in Interaction Protocols and Service Choreographies

2009

Matteo Baldoni
Dipartimento di Informatica
Università degli Studi di Torino
C.so Svizzera, 185
I-10149 Torino, Italy
baldoni@di.unito.it

Cristina Baroglio
Dipartimento di Informatica
Università degli Studi di Torino
C.so Svizzera, 185
I-10149 Torino, Italy
baroglio@di.unito.it

Amit K. Chopra
Dept. of Computer Science
North Carolina State Univ.
Raleigh, NC 27695-8206, USA
akchopra.mail@gmail.com

Nirmit Desai
IBM India Research Labs
Embassy Golf Links, Block D
Bangalore 560071, India
nirmitv@gmail.com

Viviana Patti
Dipartimento di Informatica
Università degli Studi di Torino
C.so Svizzera, 185
I-10149 Torino, Italy
patti@di.unito.it

Munindar P. Singh
Dept. of Computer Science
North Carolina State Univ.
Raleigh, NC 27695-8206, USA
singh@ncsu.edu

ABSTRACT

Many real-world applications of multiagent systems require independently designed (heterogeneous) and operated (autonomous) agents to interoperate. We consider agents who offer *business services* and collaborate in interesting business service engagements. We formalize notions of *interoperability* and *conformance*, which appropriately support agent heterogeneity and autonomy. With respect to autonomy, our approach considers the choices that each agent has, and how their choices are coordinated so that at any time one agent *leads* and its counterpart *follows*, but with initiative flu-

The accomplishment of a complex task often requires interactions among a set of parties. For instance, in a business process scenario, a seller may need to interact with a payment service and a shipper in order to support a purchase. These partners must coordinate their executions and must be able to interact with each other. There is broad agreement on the importance of describing such interactions formally. The agents community refers to such a specification as an *interaction protocol*, whereas the services community refers to it as a *choreography*. In deference to the services literature and because we do not study higher-level notions such as commit-
ments, we use the term *choreography* in this paper. A choreog

A few works on centralized runtime verification...

Runtime verification of Agent Interaction Protocols

2004

SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 8, Number 1, pp. 1–13. http://www.scpe.org

ISSN 1895-1767

SPECIFICATION AND VERIFICATION OF AGENT INTERACTION PROTOCOLS IN A LOGIC-BASED SYSTEM*

MARCO ALBERTI, FEDERICO CHESANI, DAVIDE DAOLIO, MARCO GAVANELLI, EVELINA LAMMA, PAOLA MELLO AND PAOLO TORRONI

Abstract.

A number of information systems can be described as a set of interacting entities, which must follow interaction protocols. These protocols determine the behaviour and the properties of the overall system, hence it is of the uttermost importance that the entities behave in a conformant manner.

A typical case is that of multi-agent systems, composed of a plurality of agents without a centralized control. Compliance to protocols can be hardwired in agent programs; however, this requires that only "certified" agents interact. In open systems, composed of autonomous and heterogeneous entities whose internal structure is, in general, not accessible (open agent societies being, again, a prominent example) interaction protocols should be specified in terms of the *observable* behaviour, and compliance should be verified by an external entity.

In this paper, we propose a Java-Prolog-*CHR* system for verification of compliance of computational entities to protocols specified in a logic-based formalism (*Social Integrity Constraints*). We also show the application of the formalism and the system to the specification and verification of three different scenarios: two specifications show the feasibility of our approach in the context of Multi Agent Systems (FIPA Contract-Net Protocol and Semi-Open societies), while a third specification applies to the specification of a lower level protocol (Open-Connection phase of the TCP protocol).

Runtime verification of Agent Interaction Protocols

Based on the notion of **expectation**, verified using **abductive logic programming**

Table 3.4 Integrity Constraints and Knowledge Base for the *query_ref* specification.

$$
\begin{aligned}
\mathcal{IC}: \ & \mathbf{H}(tell(A, B, query\_ref(Info), D), T) \ \wedge \ qr\_deadline(TD) \\
\rightarrow \ & \mathbf{E}(tell(B, A, inform(Info, Answer), D), T1) \ : T1 < T + TD \\
\vee \ & \mathbf{E}(tell(B, A, refuse(Info), D), T1) \ : T1 < T + TD \\
\\
& \mathbf{H}(tell(A, B, inform(Info, Answer), D), Ti) \\
\rightarrow \ & \mathbf{EN}(tell(A, B, refuse(Info), D), Tr) \\
KB: \ & qr\_deadline(10).
\end{aligned}
$$

Runtime verification of Agent Interaction Protocols

ELSEVIER

Journal of Applied Logic

Volume 5, Issue 2, June 2007, Pages 214–234

Logic-Based Agent Verification

Specifying and verifying interaction protocols in a temporal action logic

Laura Giordano[a, 1], Alberto Martelli[b, 1], Camilla Schwind[c]

Show more

doi:10.1016/j.jal.2005.12.011

Abstract

In this paper we develop a logical framework for specifying and verifying systems of communicating agents and interaction protocols. The framework is based on Dynamic Linear Time Temporal Logic (DLTL), which extends LTL by strengthening the *until* operator by indexing it with the regular programs of dynamic logic. The framework provides a simple formalization of the communicative actions in terms of their effects and preconditions and the specification of an interaction protocol by means of temporal

2007

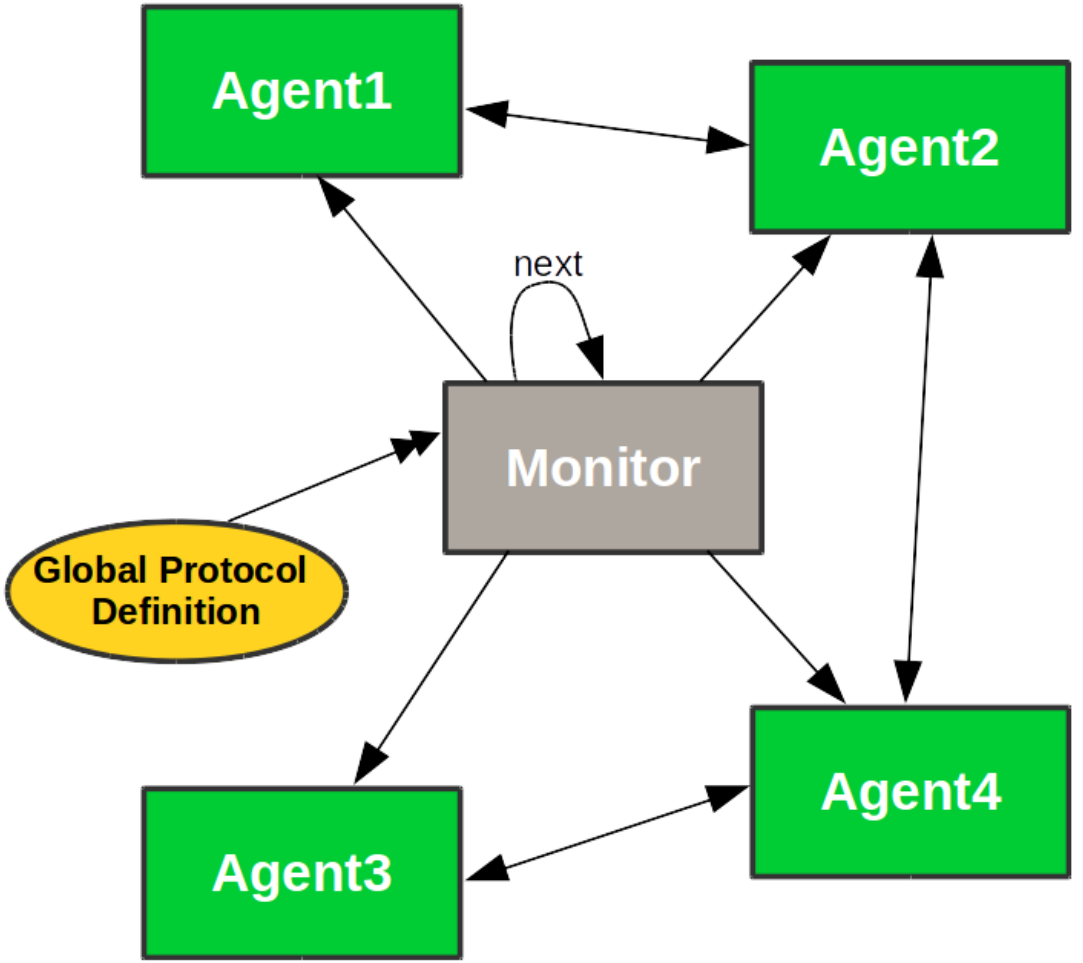
Formalism used to express protocols: Dynamic LTL (the next state modality is indexed by action).

...but... the whole trace of exchanged messages needs to be kept in memory

Part II

From centralized RV to DRV (and beyond!)

From runtime verification to self-adaptive protocol-driven behavior



Centralized monitoring

From runtime verification to self-adaptive protocol-driven behavior

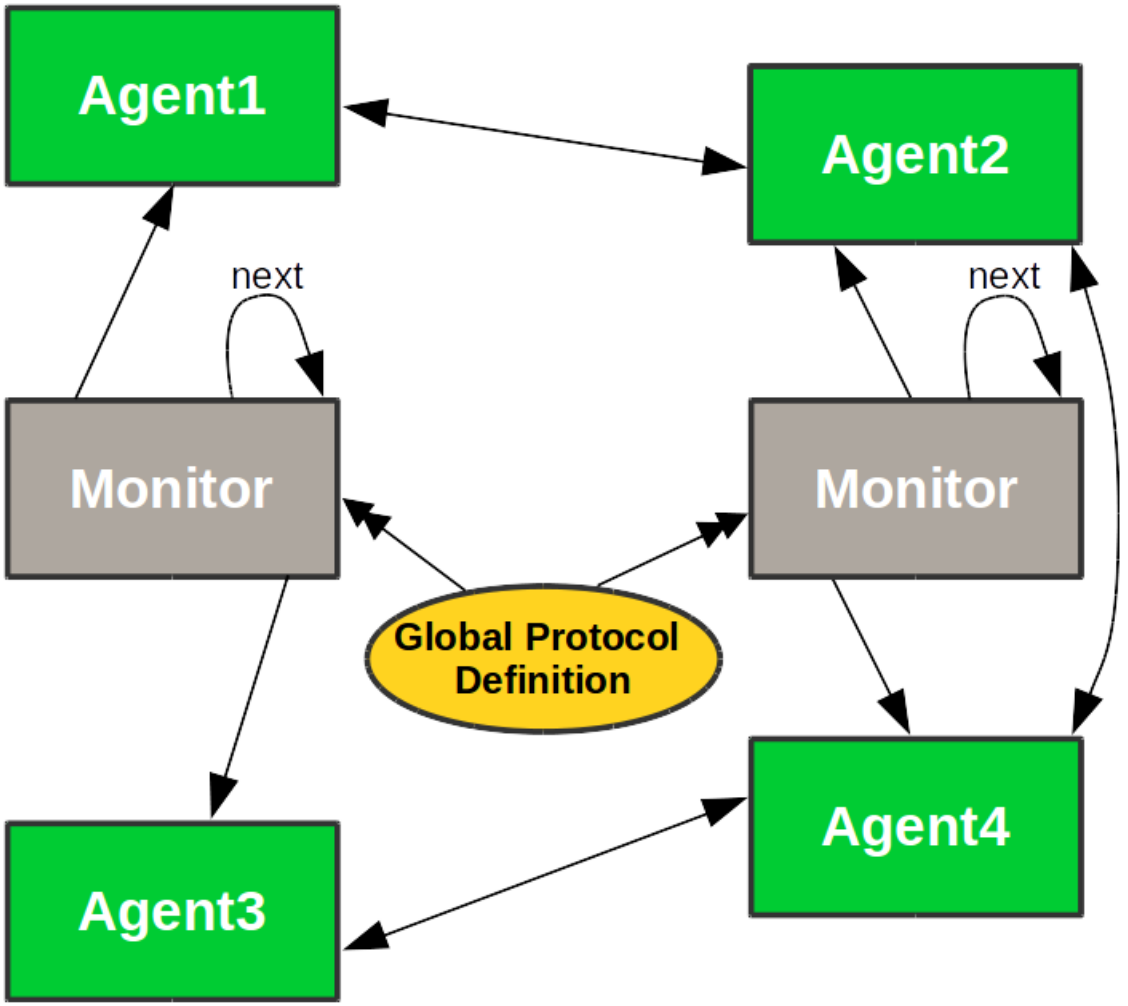
Some assumptions:

- protocols are “well formed”

- the monitor keeps track of the current state **S** of the interaction protocol (more in general, it could be any protocol involving not only communicative actions) and, as soon as it observes the event **a**, it is able to apply a “next” transition function to **S** and **a**, moving to **S'** (if possible!): **next(S, a) = S'**

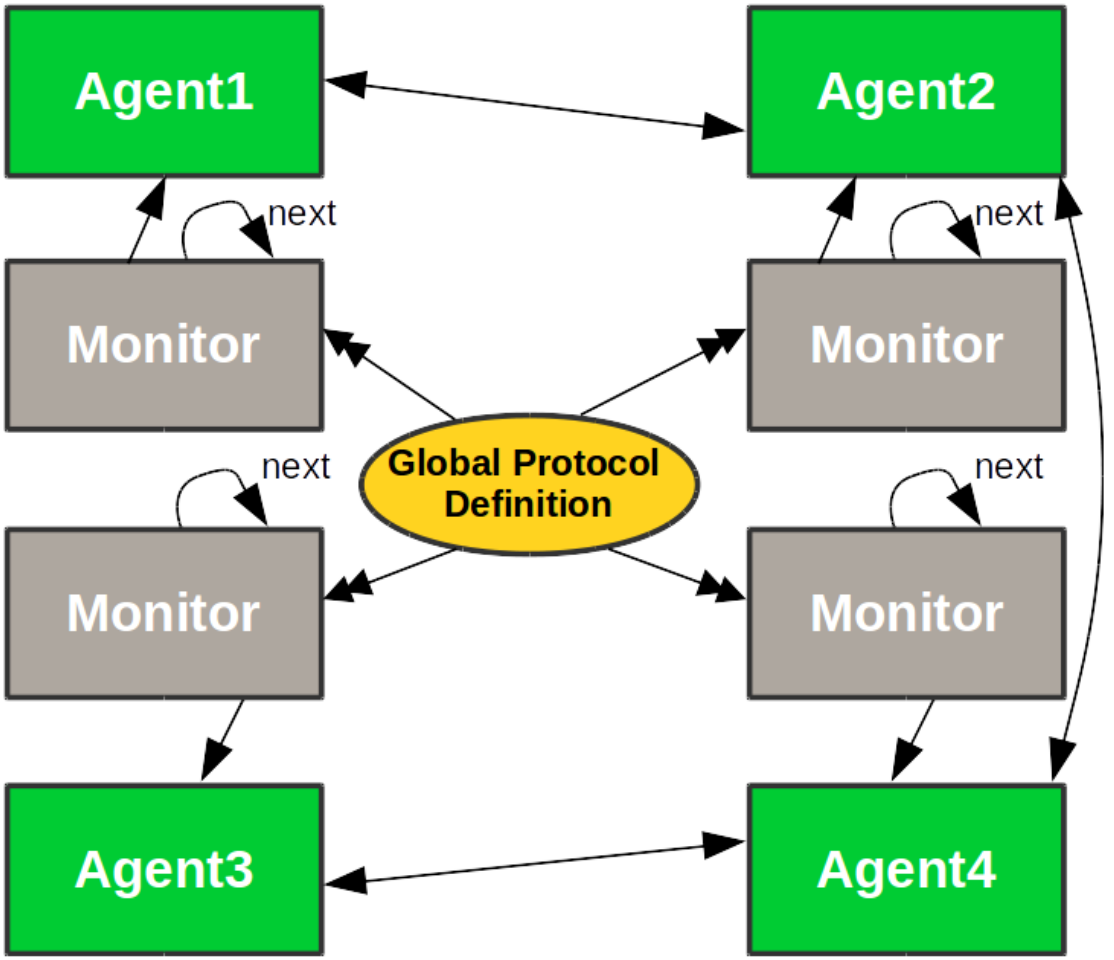
- it is possible to project any well-formed protocol **P** onto any subset of agents **{A1, A2, ..., An}**, obtaining a new protocol **P'** (in the same formalism used for representing P) where interactions involving agents **{A1, A2, ..., An}** are kept, and the others are discarded

From runtime verification to self-adaptive protocol-driven behavior



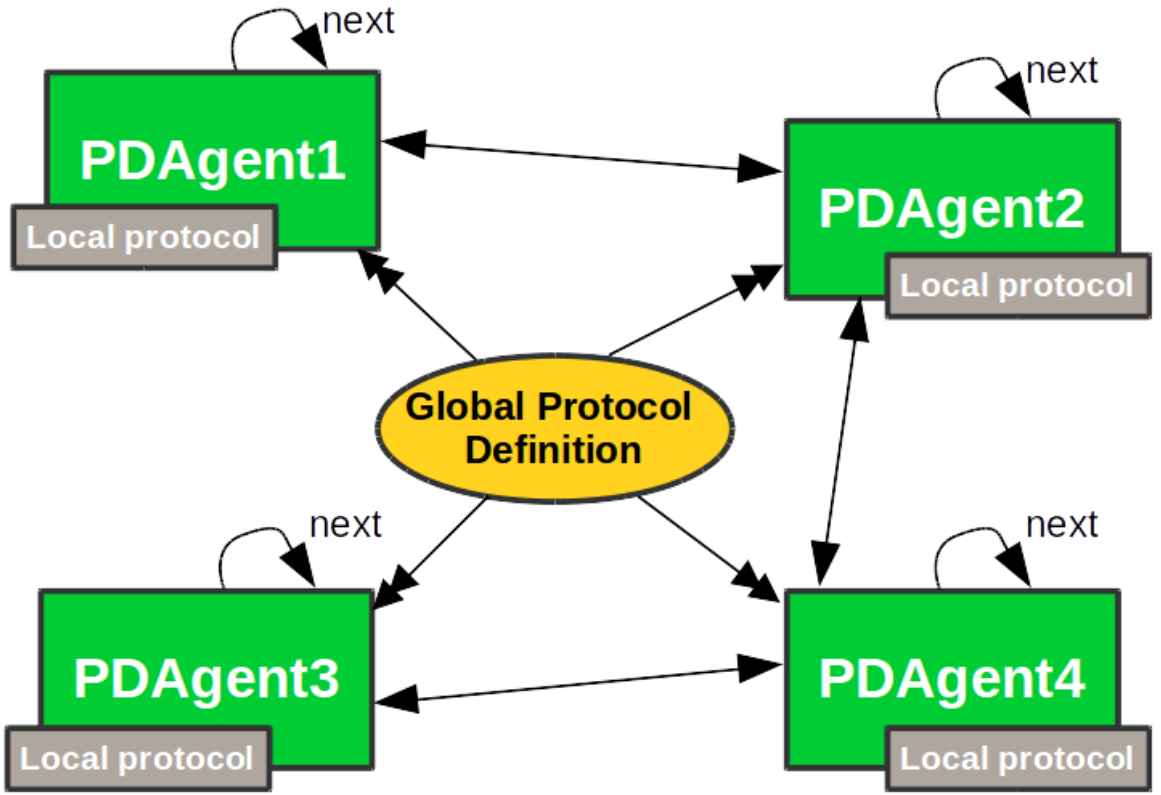
Decentralized monitoring

From runtime verification to self-adaptive protocol-driven behavior



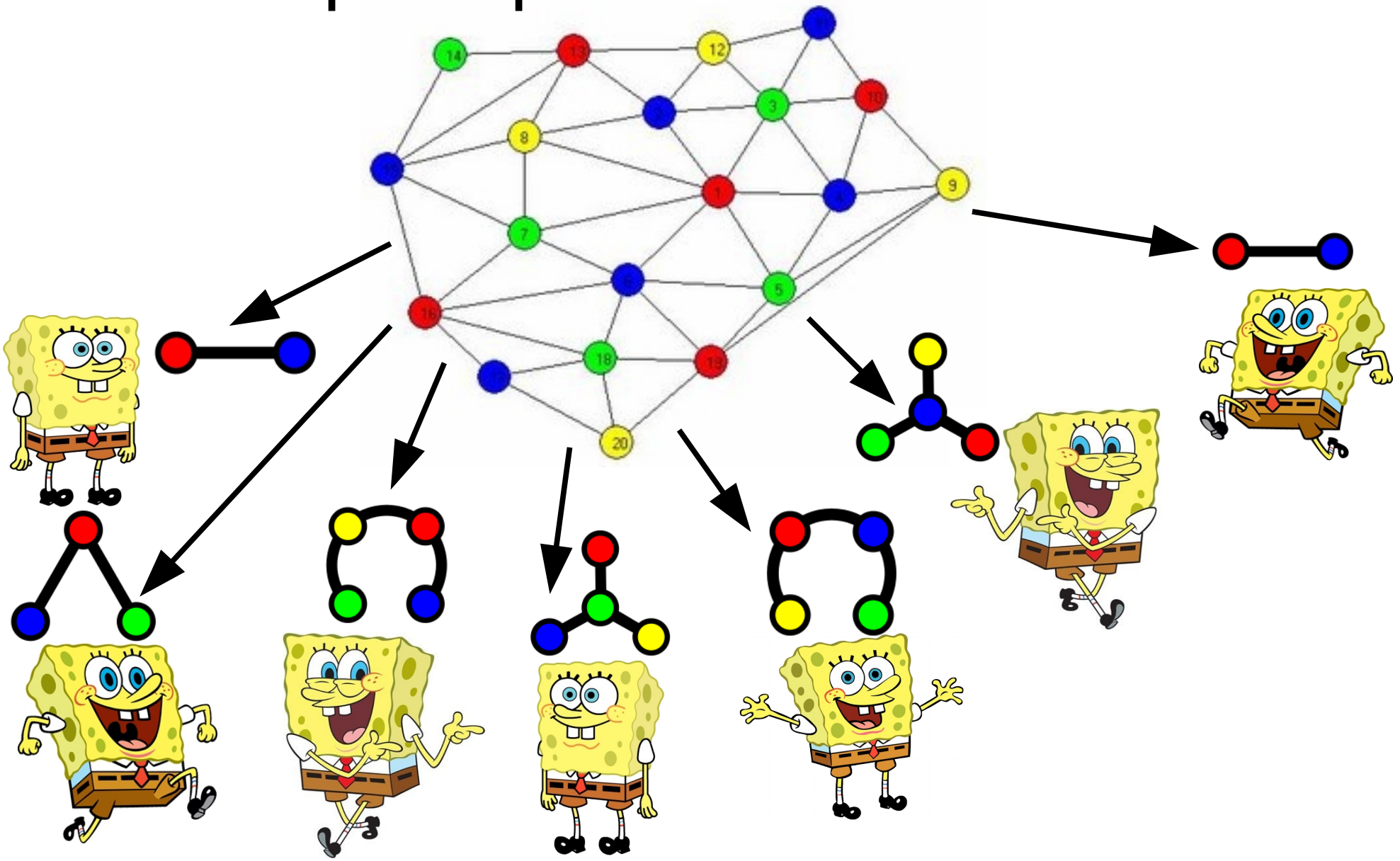
Ultra-decentralized monitoring

From runtime verification to self-adaptive protocol-driven behavior

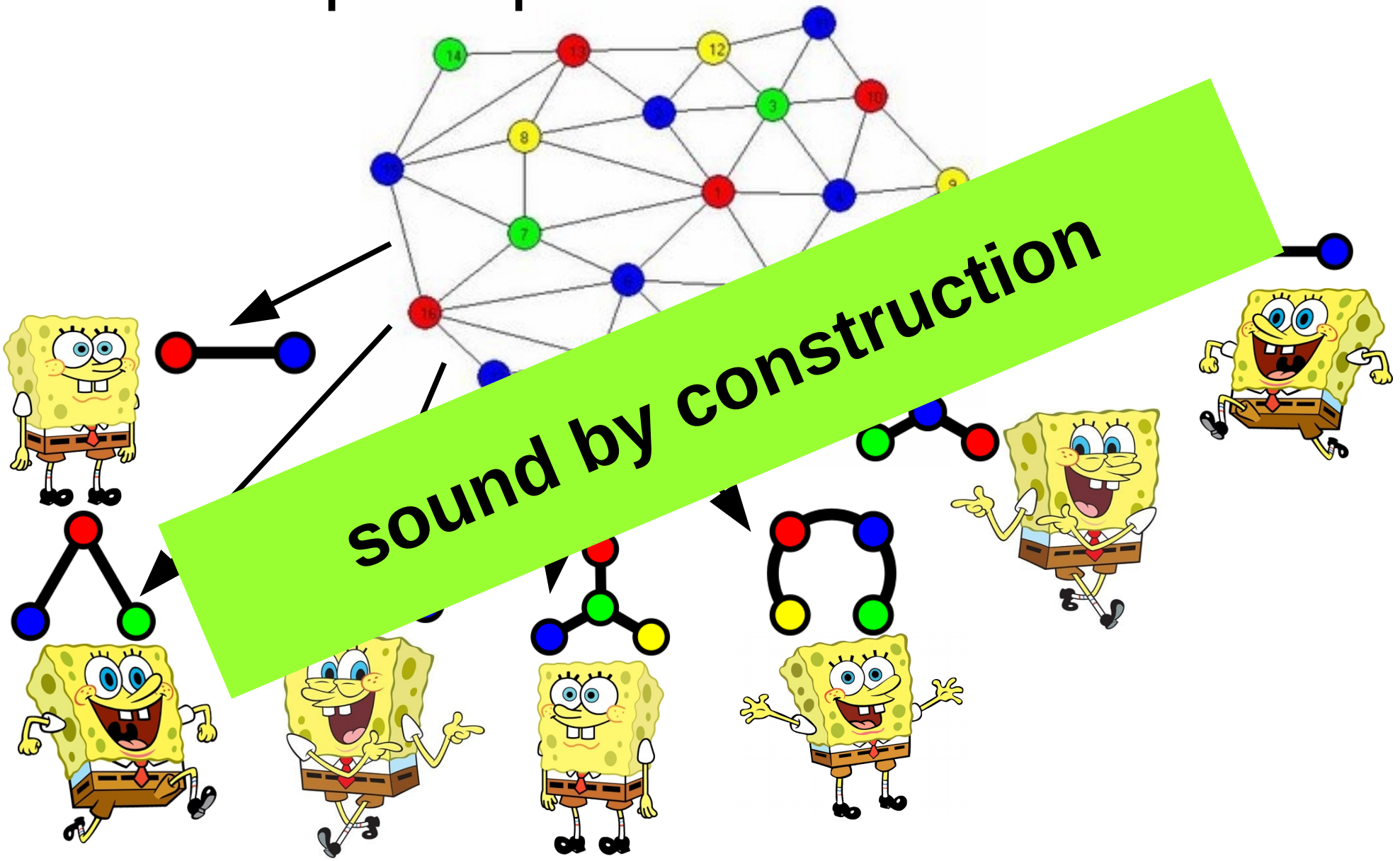


Protocol driven-agent

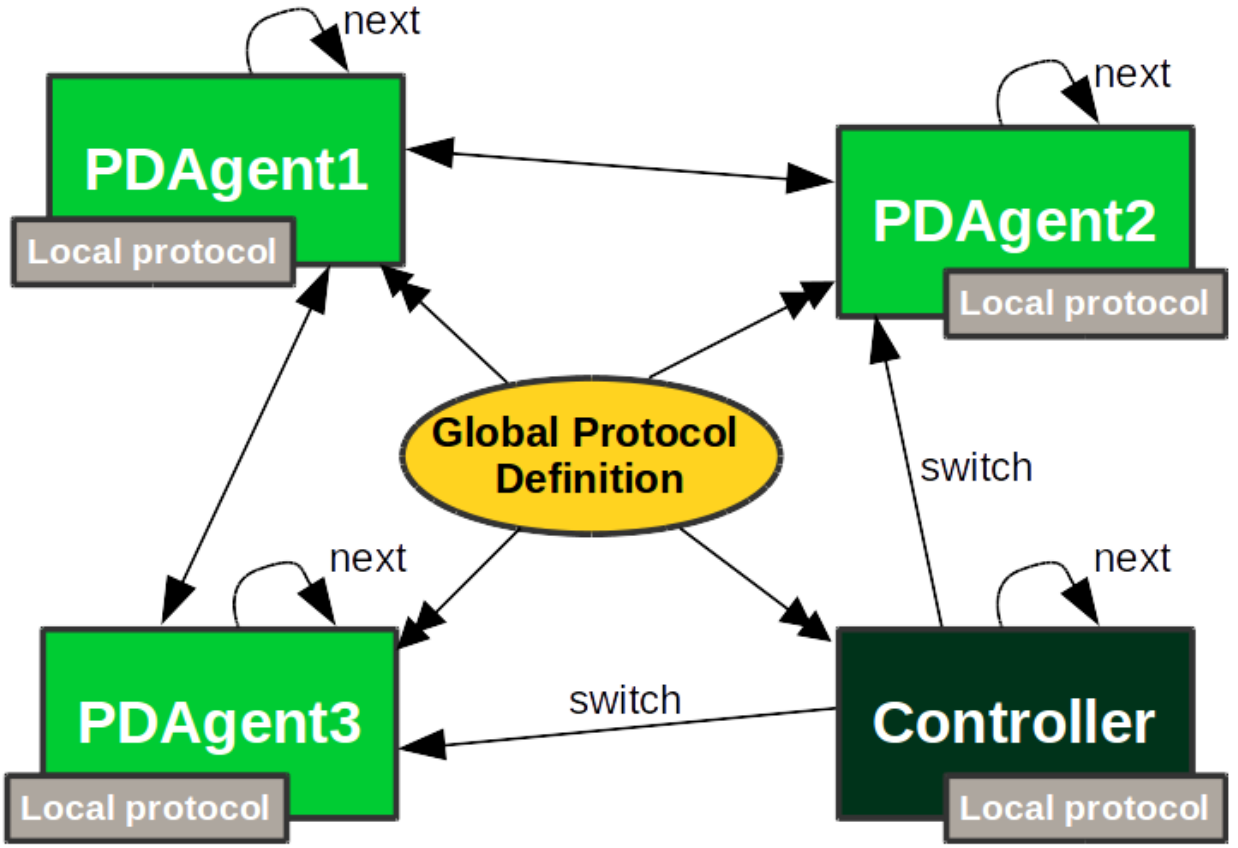
From runtime verification to self-adaptive protocol-driven behavior



From runtime verification to self-adaptive protocol-driven behavior

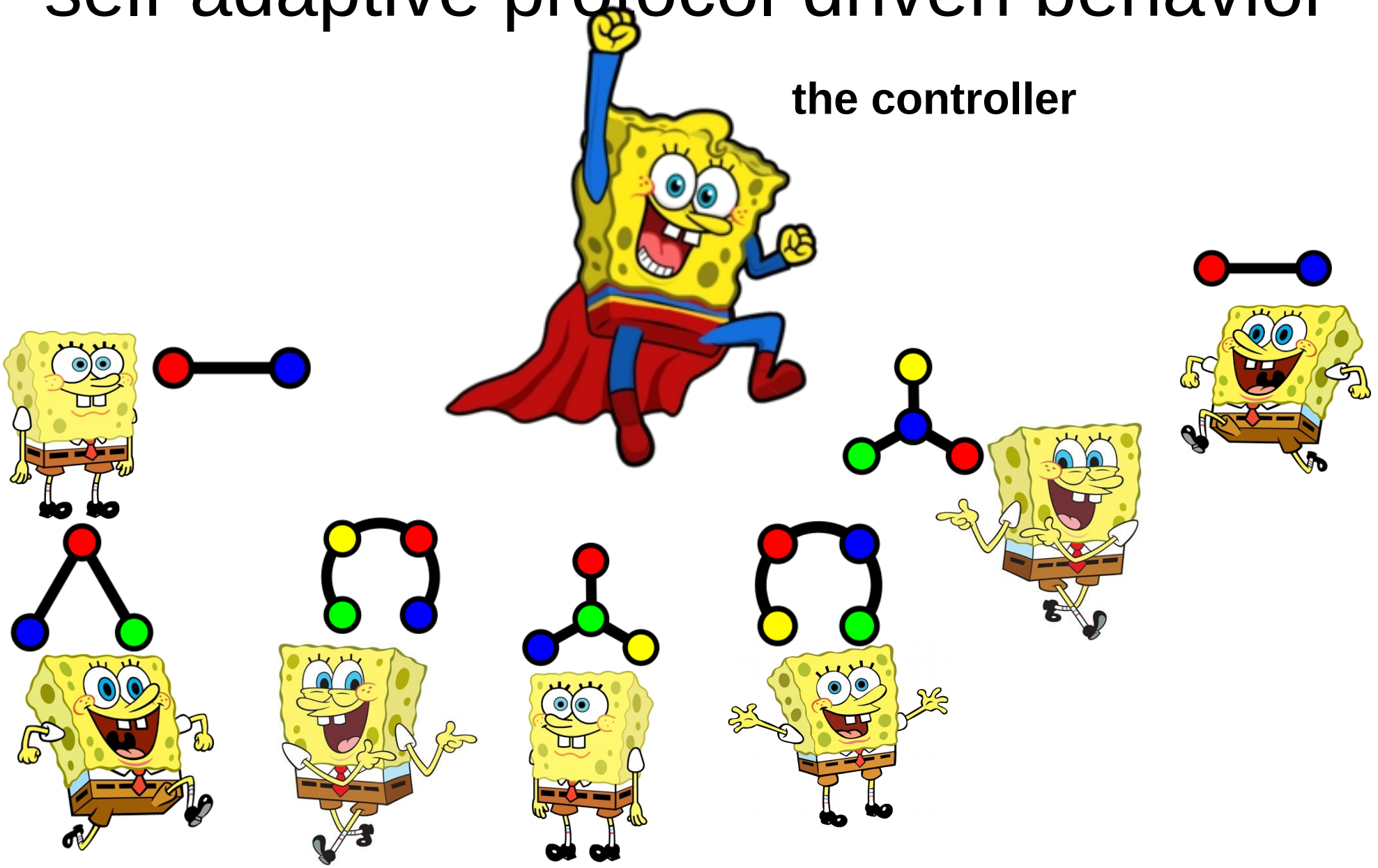


From runtime verification to self-adaptive protocol-driven behavior

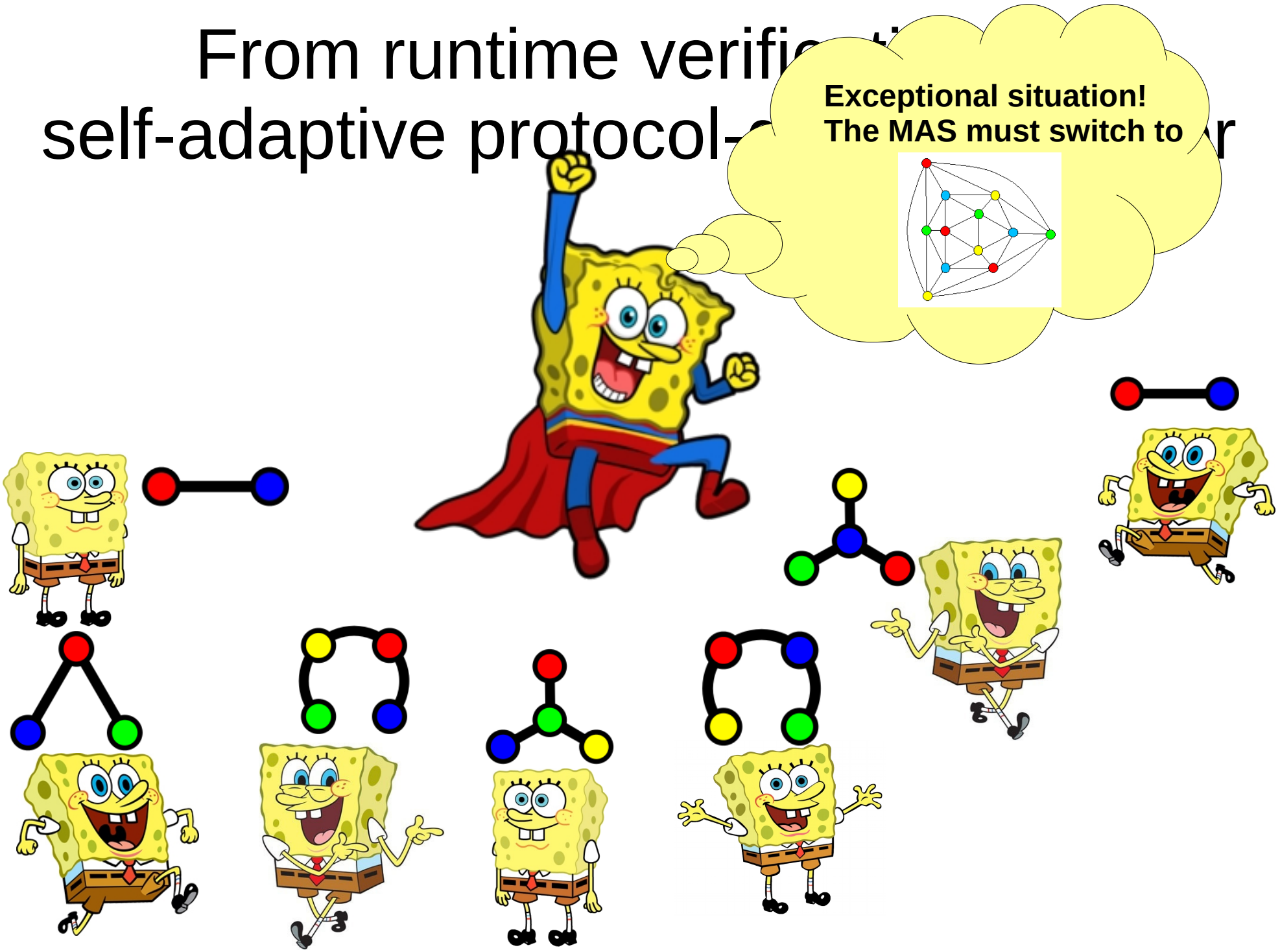


Self-adaptive protocol driven agent

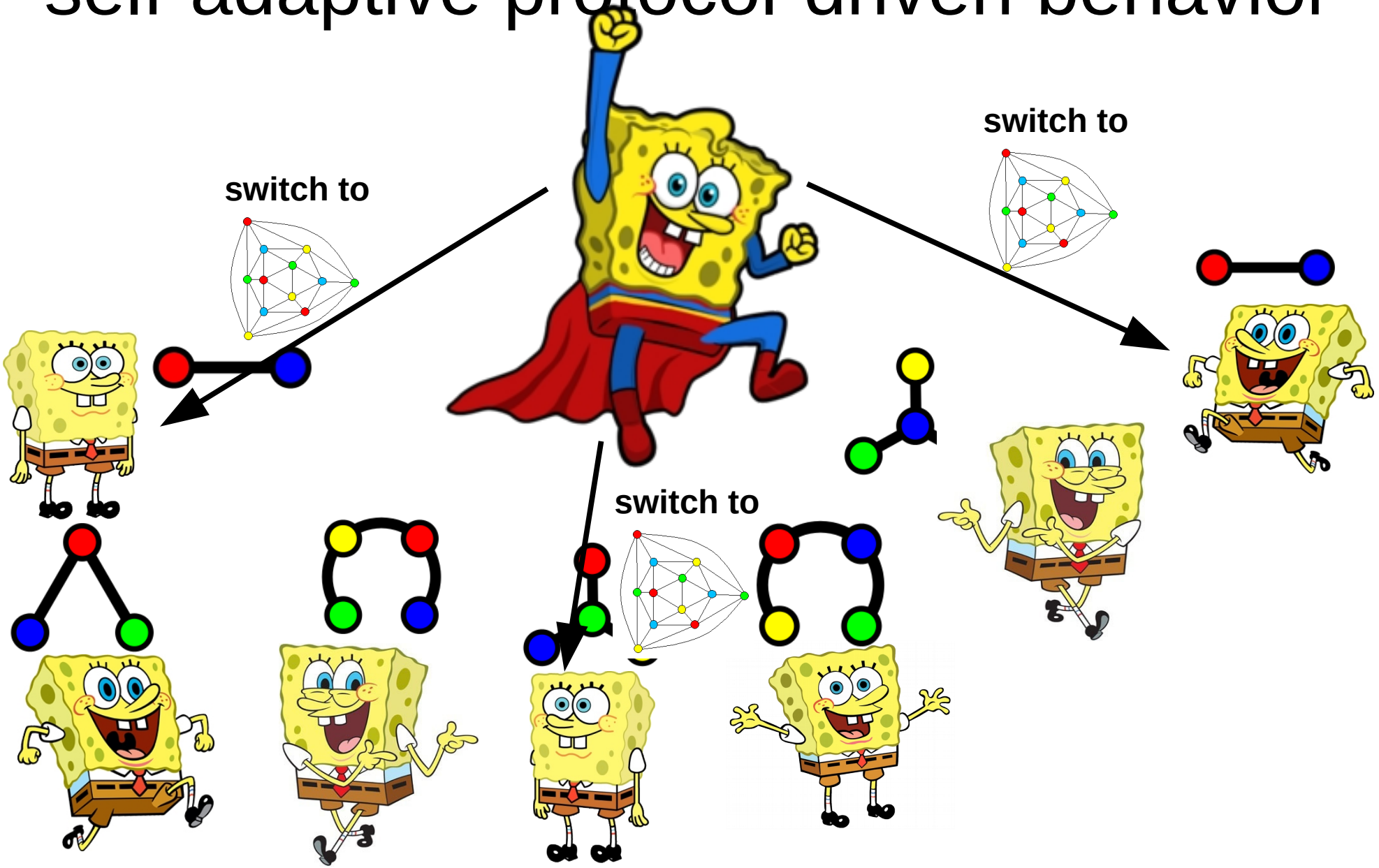
From runtime verification to self-adaptive protocol-driven behavior



From runtime verifi… self-adaptive protocol-… r

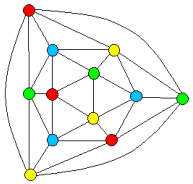


From runtime verification to self-adaptive protocol-driven behavior



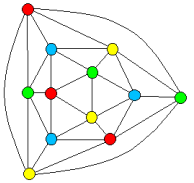
From runtime verification to self-adaptive protocol-driven behavior

project

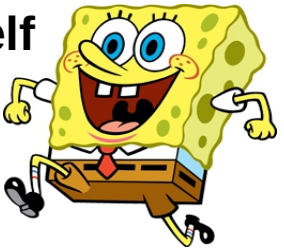


onto myself

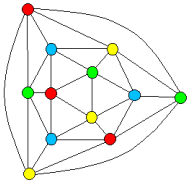
project



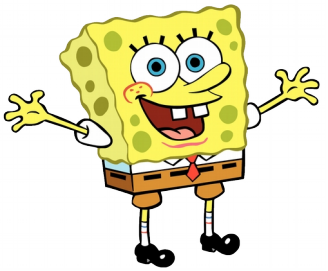
onto myself



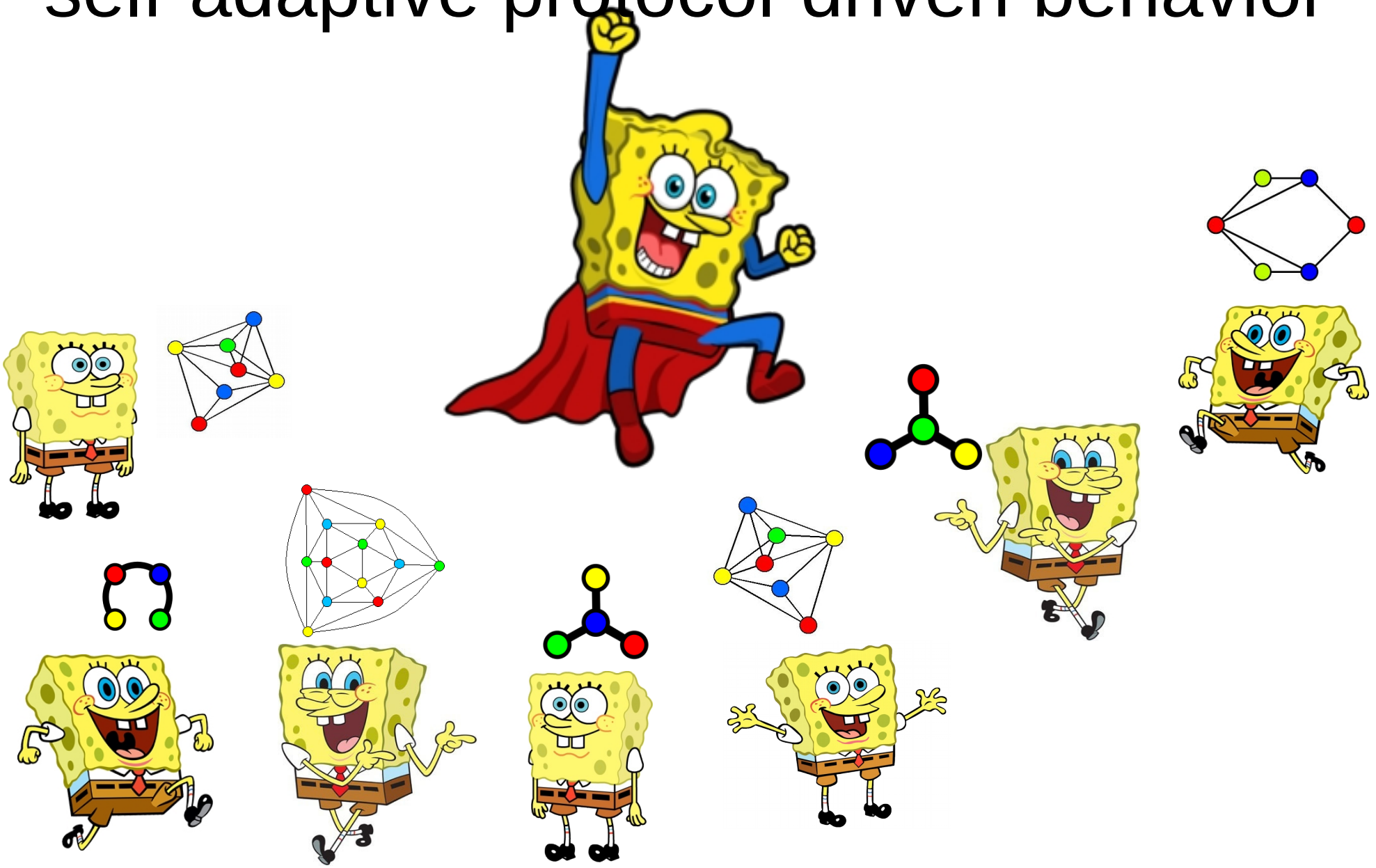
project



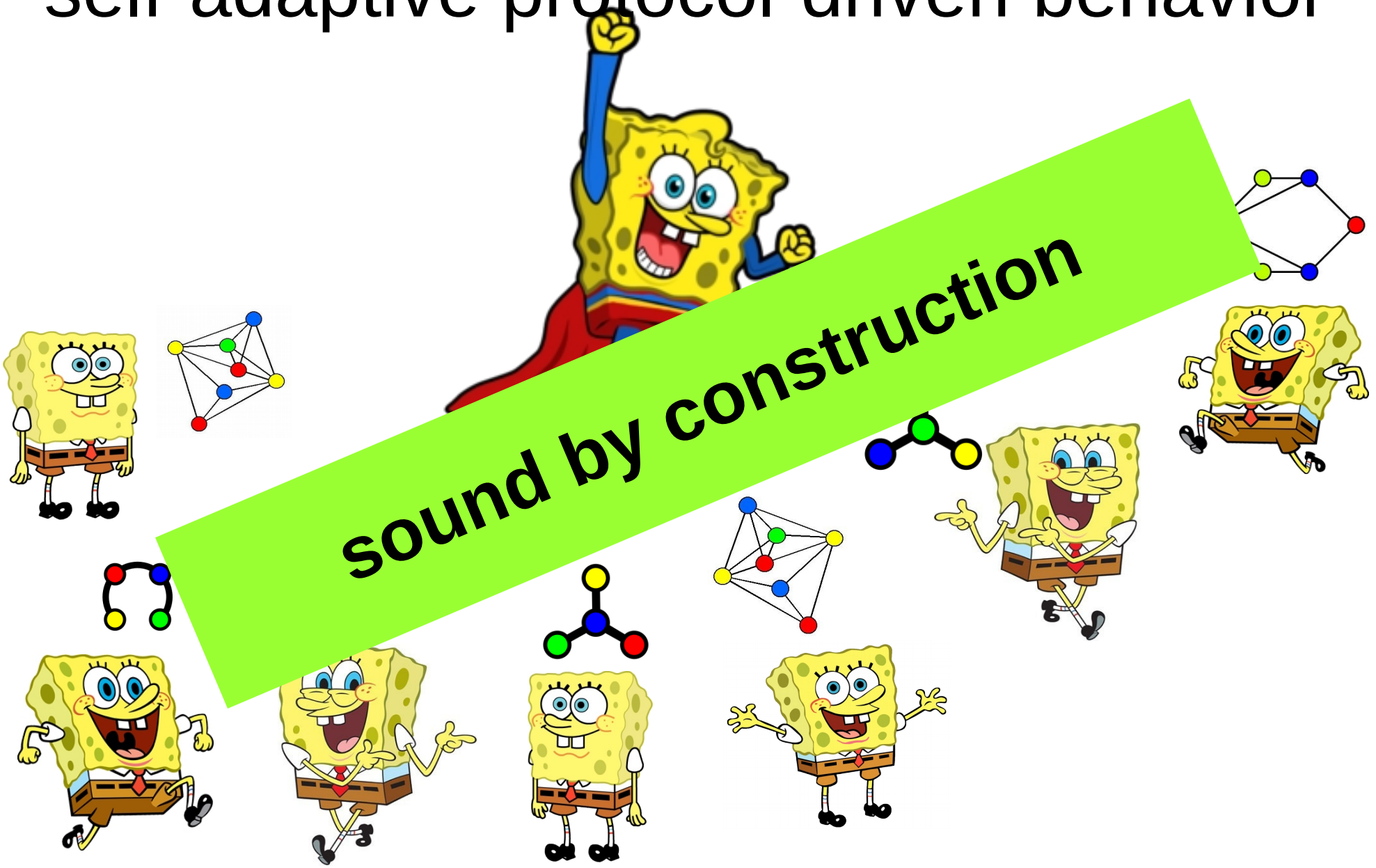
onto myself



From runtime verification to self-adaptive protocol-driven behavior



From runtime verification to self-adaptive protocol-driven behavior



Trace expressions

D. Ancona, D. Briola, A.Ferrando, V. Mascardi + ...

Trace expressions are a compact and expressive formalism which can be employed to model complex protocols based on a set of operators to denote finite and infinite traces of events.

[D. Ancona, S. Drossopoulou and V. Mascardi, DALT 2012]
[D. Ancona, M. Barbieri and V. Mascardi, SAC 2013]
[D. Ancona and V. Mascardi, TPLP 2013]
[D. Ancona, D. Briola, V. Mascardi, AI*IA 2014]
[D. Ancona, D. Briola, A. El Fallah-Seghrouchni, V. Mascardi, P. Taillibert, EMAS@AAMAS 2014]
[D. Ancona, D. Briola, V. Mascardi, IDC 2014]
[A. Ferrando, CILC 2015]
[D. Ancona, D. Briola, A. Ferrando and V. Mascardi, AAMAS 2015]
[D. Ancona, D. Briola, A. Ferrando and V. Mascardi, WOA 2015]
[D. Ancona, D. Briola, A. Ferrando and V. Mascardi, Intelligenza Artificiale 2015]
[D.Ancona, A. Ferrando and V. Mascardi, FdB60 2016]
[D.Ancona, A. Ferrando and V. Mascardi, EMAS@AAMAS 2016]

Trace expressions

Event types

msg(S, R, tell, price(Good, Price)) ∈ price_inf_msg,
for any sender S, receiver R, Good allowed good identifier, Price natural number in some range

msg(a, b, tell, msg1) ∈ msg(1)
msg(a, b, tell, msg2) ∈ msg(2)
msg(b, a, tell, ack1) ∈ ack(1)
msg(b, a, tell, ack2) ∈ ack(2)
msg(a, b, tell, msg1) ∈ msg
msg(a, b, tell, msg2) ∈ msg
msg(a, b, tell, msg1) ∈ msg_ack(1)
msg(a, b, tell, msg2) ∈ msg_ack(2)
msg(b, a, tell, ack1) ∈ msg_ack(1)
msg(b, a, tell, ack2) ∈ msg_ack(2)

Trace expressions

Syntax & (informal) semantics

• ε (empty trace), denoting the singleton set {ε} containing the empty event trace ε;
• θ:τ (prefix), denoting the set of all traces whose first event e matches the event type θ (e ∈ θ), and the remaining part is a trace of τ;
• τ1 · τ2 (concatenation), denoting the set of all traces obtained by concatenating the traces of τ1 with those of τ2;
• τ1 ∧ τ2 (intersection), denoting the intersection of the traces of τ1 and τ2 ;
• τ1 ∨ τ2 (union), denoting the union of the traces of τ1 and τ2;
• τ1 | τ2 (shuffle), denoting the set obtained by shuffling the traces in τ1 with the traces in τ2.

• θ >> τ (filter - derived operator) denoting the set of all traces contained in τ, when deprived of all events that do not match θ.

Trace expressions

Operational semantics of trace expressions

$$\text{(prefix)}\ \frac{}{\vartheta : \tau \xrightarrow{e} \tau}\ e \in \vartheta \qquad \text{(or-l)}\ \frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 \vee \tau_2 \xrightarrow{e} \tau_1'} \qquad \text{(or-r)}\ \frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \vee \tau_2 \xrightarrow{e} \tau_2'} \qquad \text{(and)}\ \frac{\tau_1 \xrightarrow{e} \tau_1' \quad \tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \wedge \tau_2 \xrightarrow{e} \tau_1' \wedge \tau_2'}$$

$$\text{(shuffle-l)}\ \frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 | \tau_2 \xrightarrow{e} \tau_1' | \tau_2} \qquad \text{(shuffle-r)}\ \frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1 | \tau_2 \xrightarrow{e} \tau_1 | \tau_2'} \qquad \text{(cat-l)}\ \frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 \cdot \tau_2 \xrightarrow{e} \tau_1' \cdot \tau_2} \qquad \text{(cat-r)}\ \frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \cdot \tau_2 \xrightarrow{e} \tau_2'}\ \varepsilon(\tau_1)$$

$$\text{(cond-t)}\ \frac{\tau \xrightarrow{e} \tau'}{\vartheta \gg \tau \xrightarrow{e} \vartheta \gg \tau'}\ e \in \vartheta \qquad \text{(cond-f)}\ \frac{}{\vartheta \gg \tau \xrightarrow{e} \vartheta \gg \tau}\ e \notin \vartheta$$

Trace expressions

Empty trace containment

$$(\varepsilon\text{-empty})\ \frac{}{\varepsilon(\varepsilon)} \qquad (\varepsilon\text{-or-l})\ \frac{\varepsilon(\tau_1)}{\varepsilon(\tau_1 \vee \tau_2)} \qquad (\varepsilon\text{-or-r})\ \frac{\varepsilon(\tau_2)}{\varepsilon(\tau_1 \vee \tau_2)} \qquad (\varepsilon\text{-shuffle})\ \frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1 | \tau_2)}$$

$$(\varepsilon\text{-cat})\ \frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1 \cdot \tau_2)} \qquad (\varepsilon\text{-and})\ \frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1 \wedge \tau_2)} \qquad (\varepsilon\text{-cond})\ \frac{\varepsilon(\tau)}{\varepsilon(\vartheta \gg \tau)}$$

Compact implementation of these transition rules, which implement the “next” function, in SWI-Prolog: basically, we needed one clause for each rule.

Trace expressions

Examples

Let us suppose that event a has type A and event b has type B

- τ = A:ε denotes {a}
- τ = A:ε v B:ε denotes {a, b}
- τ = A:B:ε denotes {ab}
- τ = A:τ denotes the set $\{a^{\omega}\}$ (coinductive interpretation of syntactic equations!)

Trace expressions

- $\tau = A{:}\tau \vee \varepsilon$ denotes the set $\{a^n \mid n > 0\} \cup \{a^\omega\}$

Non-context-free language $\{a^n b^n c^n \mid n \geq 0\}$

$$[\![a]\!] = \{a\}$$
$$[\![b]\!] = \{b\}$$
$$[\![c]\!] = \{c\}$$
$$[\![a\_or\_b]\!] = \{a, b\}$$
$$[\![b\_or\_c]\!] = \{b, c\}$$

Trace expression

$$\begin{aligned} T &= (a\_or\_b \gg AB) \wedge (b\_or\_c \gg BC) \\ AB &= \epsilon \vee (a{:}(AB \cdot (b{:}\epsilon))) \\ BC &= \epsilon \vee (b{:}(BC \cdot (c{:}\epsilon))) \end{aligned}$$

Examples

$aabbcc \in [\![T]\!]$

$aabcc \notin [\![T]\!]$

Trace expressions

Alternating bit protocol [DeniélouYoshida12]

$msg(i)$: A sends to B message of kind i ($i \in \{1,2\}$)

$ack(i)$: B sends to A ack of message of kind i ($i \in \{1,2\}$)

msg: $msg(1)$ or $msg(2)$

$msg\_ack(i)$: $msg(i)$ or $ack(i)$

Protocol specification

$msg(1)^n < msg(2)^n < msg(1)^{n+1}$ for all $n \in \mathbb{N}$

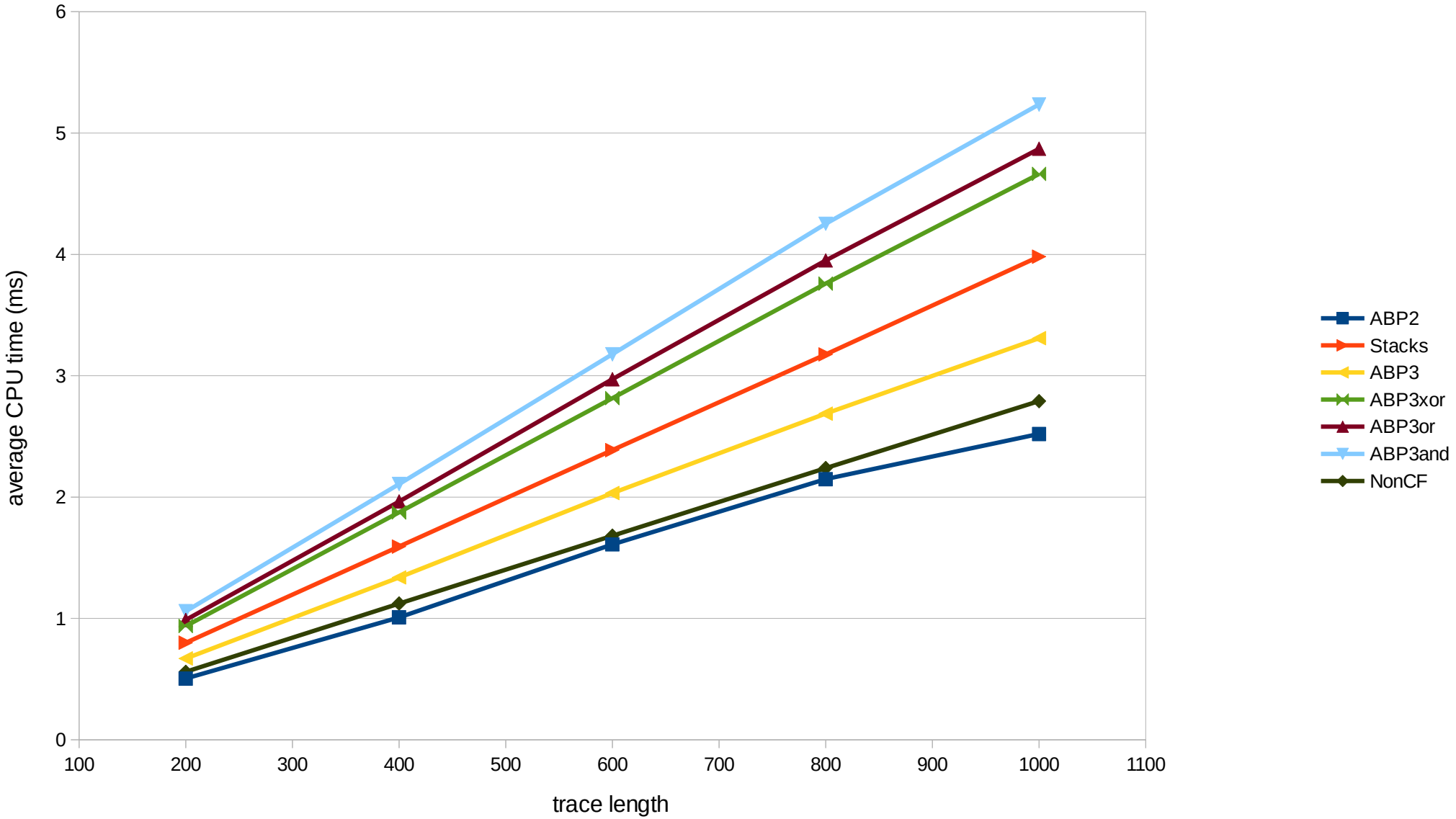
$msg(1)^n < ack(1)^n < msg(1)^{n+1}$ for all $n \in \mathbb{N}$

$msg(2)^n < ack(2)^n < msg(2)^{n+1}$ for all $n \in \mathbb{N}$

Trace expression

$$
\begin{array}{rcl}
AltBit & = & (msg \gg MM) \wedge (msg\_ack(1) \gg MA_1) \wedge (msg\_ack(2) \gg MA_2) \\
MM & = & msg(1) : msg(2) : MM \\
MA_i & = & msg(i) : ack(i) : MA_i \quad (i \in \{1,2\})
\end{array}
$$

Trace expressions



Implementation

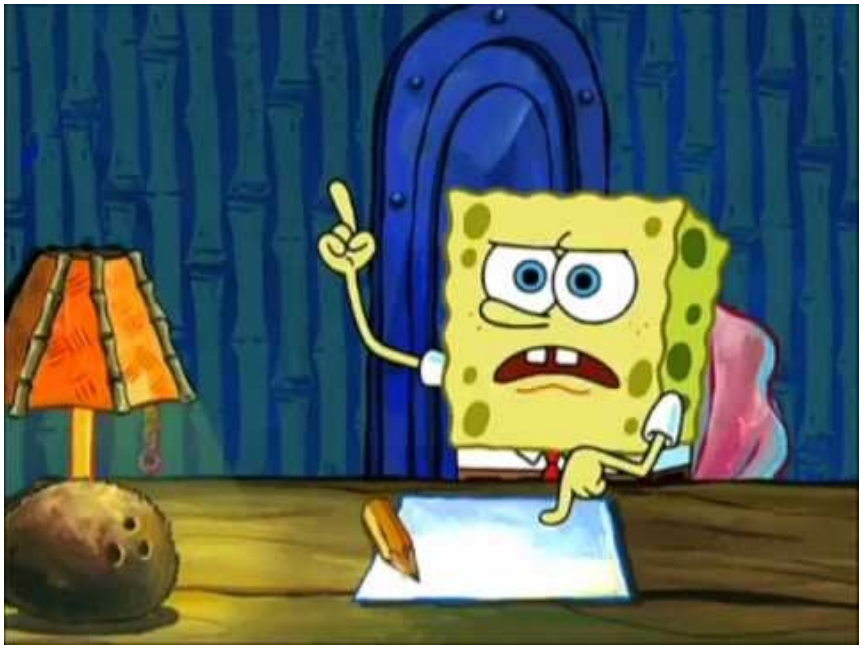
The interpreter for decentralized monitoring and protocol-driven agents using trace expressions has been implemented using SWI-Prolog and runs on top of

SWI Prolog

SWI Prolog

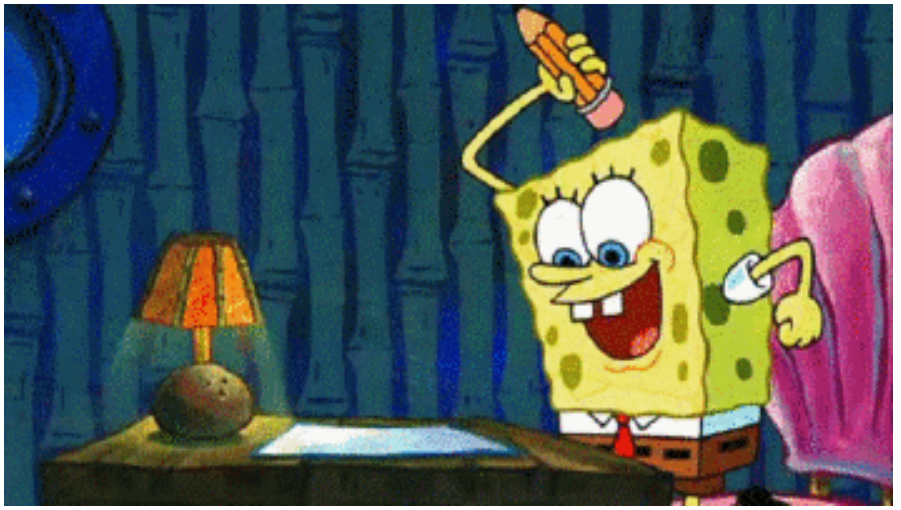
Conclusions

- MASs are Distributed Systems
- Agents must interact (and, in general, behave) following some well known protocol: this calls for techniques to verify that they actually do that. A lot of work on a priori verification.
- The strong agent notion has been traditionally modeled using the same logics at the basis of model checking. A lot of work on model checking agent programs.

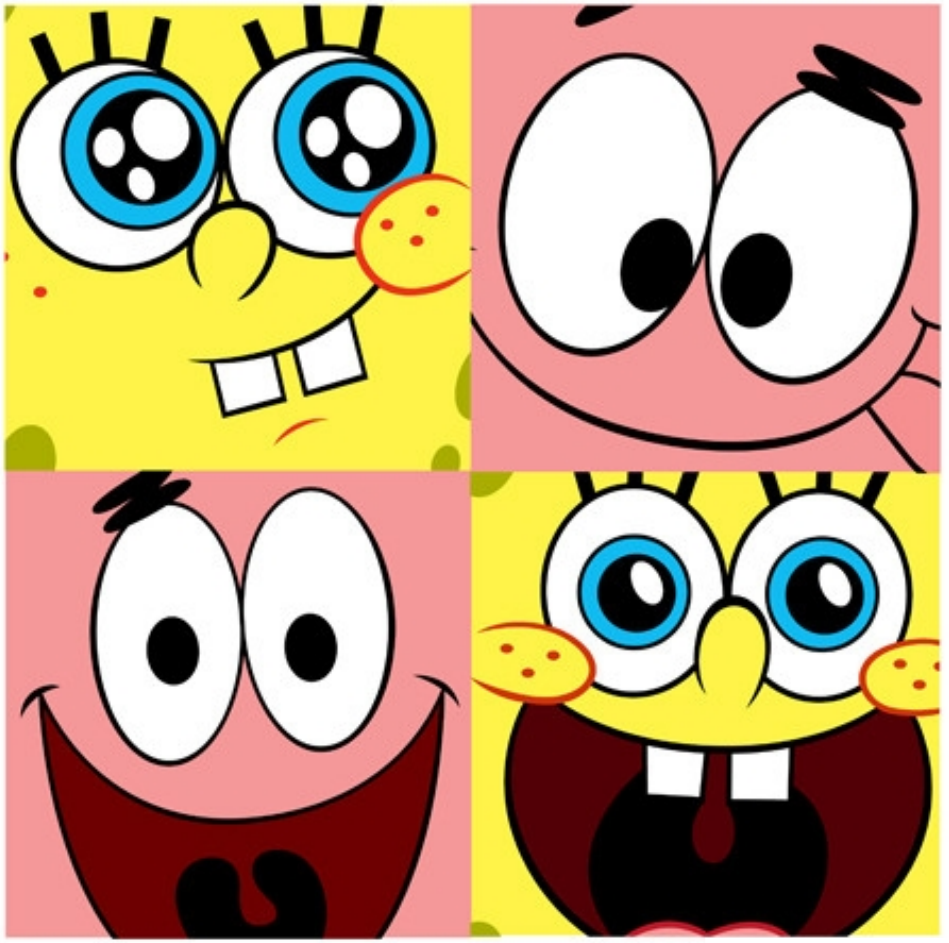


Conclusions

- Little work on MAS runtime verification.
- Just initial works on MAS decentralized runtime verification.
- How to decide which subsets of the MAS should be grouped together to be monitored by the same monitor is still an open issue...
- ...but if we associate one monitor with each agent, we can then “push the monitor inside the agent” and obtain a “protocol driven agent”
- How to make protocols parametric is another open issue (looking forward this afternoon talks to learn more about it!)



The tutorial is over...



...thank you for your attention!
Questions?

Some pointers to works on trace expressions

2016

Davide Ancona, Angelo Ferrando, Viviana Mascardi: Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification. Theory and Practice of Formal Methods 2016: 47-64

2015

Davide Ancona, Daniela Briola, Angelo Ferrando, Viviana Mascardi: Runtime verification of fail-uncontrolled and ambient intelligence systems: A uniform approach. Intelligenza Artificiale 9(2): 131-148 (2015)

Davide Ancona, Daniela Briola, Angelo Ferrando, Viviana Mascardi: Global Protocols as First Class Entities for Self-Adaptive Agents. AAMAS 2015: 1019-1029

Some pointers to works on trace expressions

2015

Davide Ancona, Daniela Briola, Viviana Mascardi: Protocols with Exceptions, Timeouts, and Handlers: A Uniform Framework for Monitoring Fail-Uncontrolled and Ambient Intelligence Systems. WOA 2015: 65-75

2014

Davide Ancona, Daniela Briola, Amal El Fallah-Seghrouchni, Viviana Mascardi, Patrick Taillibert: Efficient Verification of MASs with Projections. EMAS@AAMAS 2014: 246-270

Daniela Briola, Viviana Mascardi, Davide Ancona: Distributed Runtime Verification of JADE Multiagent Systems. IDC 2014: 81-91

Some pointers to works on trace expressions

2013

Viviana Mascardi, Davide Ancona: Attribute Global Types for Dynamic Checking of Protocols in Logic-based Multiagent Systems. TPLP 13(4-5-Online-Supplement) (2013)

Viviana Mascardi, Daniela Briola, Davide Ancona: On the Expressiveness of Attribute Global Types: The Formalization of a Real Multiagent System Protocol. AI*IA 2013: 300-311

Davide Ancona, Matteo Barbieri, Viviana Mascardi: Constrained global types for dynamic checking of protocol conformance in multi-agent systems. SAC 2013: 1377-1379

Some pointers to works on trace expressions

2012

Davide Ancona, Sophia Drossopoulou, Viviana Mascardi: Automatic Generation of Self-monitoring MASs from Multiparty Global Session Types in Jason. DALT 2012: 76-95

.

Two properties that our trace expressions should respect

Contractiveness

Definition 1. A trace expression τ is contractive if all its infinite paths from the root contain the prefix operator.

In contractive trace expressions all recursive subexpressions must be guarded by the prefix operator; for instance, the trace expression defined by T1 = (epsilon ∨(θ:T1)) is contractive: its infinite path contains infinite occurrences of ∨, but also of the : operator; conversely, the trace expression T2 = (epsilon ∨ ((T2 |T2)∨(T2 ·T2))) is not contractive.

Trivially, every trace expression corresponding to a finite tree (that is, a non cyclic term) is contractive.

For all contractive trace expressions, any path from their root must always reach either a or a : node in a finite number of steps.

Determinism

Deterministic trace expressions. There are trace expressions τ for which the problem of word recognition is less efficient because of non determinism. Non determinism originates from the union, shuffle, and concatenation operators, because for each of them two possibly overlapping transition rules are defined. We only consider deterministic trace expressions.

Definition 3. Let τ be a trace expression; τ is deterministic if for all finite event traces σ, if $\tau \xrightarrow{\sigma} \tau'$ and $\tau \xrightarrow{\sigma} \tau''$ are valid, then [[τ']] = [[τ'']] .