A Multi-Agent Framework for Dependable Adaptation of Evolving System Architectures

Kenneth Johnson[1], Roopak Sinha[1], Radu Calinescu[2], and Ji Ruan[1]

[1] *School of Engineering, Computer and Mathematical Sciences Auckland University of Technology, Auckland, New Zealand*
[2] *Department of Computer Science, University of York, York, United Kingdom*

Abstract

We present a multi-agent framework for the formal verification of component-based systems after changes such as addition, removal and modification of components. The core of our approach is an Agent Verification Engine (AVE) that constructs evolvable Belief-Desire-Intention (BDI) agents to coordinate and plan the re-verification of component models after system changes. The engine provides BDI-agents with existing techniques for the compositional verification of component-based systems. We illustrate this integration for Satisfiability Modulo Theories (SMT) constraint analysis and demonstrate our framework on requirements arising from industrial control systems.

1 Introduction

Component-based software systems are increasingly common, and include business- and safety-critical systems from domains as diverse as healthcare, transportation and finance [1]. Critical systems are expected to be reliable since downtime often results in a decrease of revenue or unavailability of essential services. Formal verification techniques such as model checking or automated theorem proving can be used to provide irrefutable proof of a system's compliance to requirements by analysing mathematical models of the system against properties derived from the requirements.

Compositional verification techniques [2, 3] decompose the verification of a large, monolithic system model into a sequence of small verification steps that are applied to the components of the system, thus enabling formal verification of much larger systems. However, these approaches do not take into account the system's runtime environment, where components are frequently added and updated or may unexpectedly fail. Agent-based modelling is widely used in safety or business critical systems as a means for intelligent adaptation in response to planned and unplanned runtime changes. This approach is well suited to adapt large scale cloud deployed systems [4, 5] and industrial manufacturing applications [6, 7] whose compliance to quality-of-service (QoS) requirements must be maintained during runtime.

In this paper, we introduce the *Agent Verification Engine* (AVE) which constructs agents to perceive, react, and adapt to runtime changes of a component-

based system. These agents are based on the *Belief-Desire-Intention* (BDI) architecture [8], in which agents operate in terms of motivation and beliefs. BDI-agents have a formal basis in logic to formalize an agent's decision making in an attempt to achieve its goals [9]. BDI-agents are constructed by AVE according to the system's architecture and observe components to detect changes. When changes happen, agents perform *verification tasks* to verify the model of components against properties to determine compliance with system requirements.

The main contributions of this paper are (a) an algebraic specification of an agent verification engine based on BDI-agents, (b) the integration of a satisfiability modulo theories (SMT) solver to provide agents with verification capabilities, (c) an application of the engine to a real-world industrial control software case study, and (d) a prototype implementation of the agent verification engine using Jason AgentSpeak [10].

The rest of this paper is organised as follows. Section 2 provides key mathematical preliminaries. Section 3 formulates the agent verification engine algebraically and describes evolvable BDI-agent plans. Section 4 integrates an existing SMT-based compositional verification approach into the engine. Section 5 applies the engine to analyse an industrial item sorting system. Section 6 shows the effectiveness and scalability of our approach. Section 7 discusses related work and Section 8 presents concluding remarks.

2 Preliminaries

Assuming readers to have basic understanding of universal algebra [11] and BDI-agent architectures [8, 10], we define the following key tools and notations.

Signatures and Terms

A signature Σ consists of a sort s and a finite number of operation symbols $f : s^n \to s$ for $n \geq 0$. Symbols of the form $c \to S$ are signature constants. A signature Σ contains the sort $Bool$ with constants $true, false \to Bool$ and names the standard logical connectives $\wedge$ and $\neg$. Operation symbols of the form $b : s^n \to Bool$ are Σ-*predicates*. Σ-terms are defined by the rules

$$t ::= c_1 \mid \ldots \mid c_m \mid z \mid f_1(t_1, \ldots, t_{m_1}) \mid \ldots \mid f_n(t_1, \ldots, t_{m_n})$$

for constants $c_1, \ldots, c_m$, variable z from the set Z, operation symbols $f_1, \ldots, f_n$ and Σ-terms t_{i_j}.

The AgentSpeak Language

AgentSpeak [8] is an agent-oriented programming based on logic programming and the BDI architecture for autonomous agents. BDI-agents operate within an environment and receive continuous input *perceptions*. Agents respond by performing actions that affect the environment. The *beliefs*, *desires* and *intentions* of an agent correspond with the agent's informational, motivational and decision making components that comprise its mental state.

Beliefs: An agent's belief is constructed from predicate terms as follows. If $\mathbf{t} = t_1, \ldots, t_n$ is a tuple of Σ-terms then $b(\mathbf{t})$ is a belief atom for the predicate

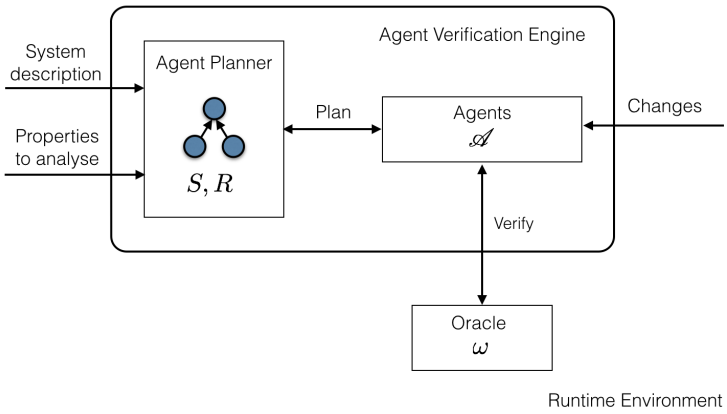

Figure 1: Agent Verification Engine (AVE) workflow

symbol b. If $b(\mathbf{t})$ and $c(\mathbf{t})$ are belief atoms then $\neg b(\mathbf{t})$, $b(\mathbf{t}) \wedge c(\mathbf{t})$ are beliefs. Beliefs represent the information available to the agent and are stored in its *belief base* $\mathscr{B}$.

Goals: If g is a predicate symbol and $\mathbf{t} = t_1, \ldots, t_n$ a tuple of Σ-terms then $!g(\mathbf{t})$ is an *achievement goal* an agent aims to achieve and $?g(\mathbf{t})$ is a *test goal* that tests if $g(\mathbf{t})$ is a true belief. Acquiring a new belief or goal creates a *trigger event* within an agent. If $b(\mathbf{t})$ is a belief atom then $+b(\mathbf{t})$, $-b(\mathbf{t})$ are triggering events corresponding to the addition and removal of beliefs. Similarly, $+!g(\mathbf{t})$, $-!g(\mathbf{t})$ and $+?g(\mathbf{t}), -?g(\mathbf{t})$ correspond to the addition and removal of achievement and test goals, respectively.

3 Agent Verification Engine

This section algebraically specifies the *Agent Verification Engine (AVE)*. The engine's workflow is depicted in Figure 1 and comprises the agent planner, that reads an algebraic system specification S and requirements R, a set $\mathscr{A}$ of BDI-agents constructed by the agent planner to verify components of S and an oracle ω. The workflow has two phases. In the *agent construction phase*, the agent planner constructs a set $\mathscr{A} = \{\alpha_{t_1}, \ldots, \alpha_{t_n}\}$ of BDI-agents for components $t_1, \ldots, t_n$ of the system S. Each agent is constructed with a plan library to determine compliance of its component with the requirements given by R. The agent invokes an external *oracle* ω which represents a compositional verification *service* to determine the component's satisfaction of requirements. In the *agent runtime phase*, an agent perceives localised component changes t' in the runtime environment and notifies the agent planner. The agent planner transforms S by performing the corresponding term substitution $S' = S[t'/t]$ of the agent's component term t with the new component term t' to obtain S'. The agent planner uses t' to update the agent's beliefs and plan library.

3.1 The Oracle

Let $\mathscr{M}$ be a set of component models, $\mathscr{P}$ a set of properties and $\mathscr{V}$ a set of values. The *oracle* is an automated process that accepts as input a verification task (m, g, a) from the set $W = \mathscr{M} \times \mathscr{P} \times [\mathscr{P} \to \mathscr{V}]$ and is modelled mathematically by the total function $\omega : W \to \mathscr{V}$ such that

$$\begin{aligned} \omega(m, g, a) = {} & \text{the value } v \text{ obtained from the process} \\ & \text{of verifying property } g \text{ on model } m \\ & \text{under assumptions of } a \end{aligned} \tag{1}$$

for model $m \in \mathscr{M}$, $g \in \mathscr{P}$ and $v \in \mathscr{V}$ and the map $a : \mathscr{P} \to \mathscr{V}$ of properties to verification values. The oracle is programmed to always terminate, potentially due to failure. If a failure occurs, then a special unverified value $\mathbf{u}$ is returned by the service. We extend the set of values to include this element by setting $\mathscr{V} = \mathscr{V} \cup \{\mathbf{u}\}$.

Our choice of notation in (1) generalises *assume-guarantee* [12, 13] model checking approaches to include decomposition techniques used in formal analysis of component-based systems. In assume-guarantee reasoning, a monolithic model m is decomposed into smaller, more manageable models $m_1, \ldots, m_q$. Each model m_i is associated with a property $a_i \in \mathscr{P}$, formalising requirements of the component. Component-wise verification is performed in a sequence of steps whereby the verification of property a_i on model m_i obtains a verification value $v_i \in \mathscr{V}$. In symbols, we write $m_i, v_i \models a_i$, $1 \leq i \leq q$. We denote the list of assumptions and verification results obtained by component-wise verification as $\mathbf{a}$. Using the assume-guarantee approach, the system requirements formalised as the property $g \in \mathscr{P}$ for the monolithic model m has verification result v under the assumptions $\mathbf{a}$. In symbols, we write $m, v \models g$ (m can be omitted if clear from the context).

3.2 Agent Construction Phase

We give an algebraic specification of component-based systems and use their inductive properties to derive the agent planner's construction of the agent set $\mathscr{A}$.

Specifying Component-Based Systems

A component signature Σ is a finite set C of component sorts that represent parts of a system and a set $\{f_1, \ldots, f_n\}$ of operations to be performed on components. By choosing some basic components $c_1, \ldots, c_m$, e.g. constants in Σ and applying a sequence of operations, we form a component-based system as a high-level syntactic *component term* S. The term defines the hierarchical structure of a component-based system by expressing the sequence of operations carried out in the construction of the system and specifies the order in which they are applied. Let the set of all component terms over the signature Σ be denoted as $C(\Sigma)$. For example, the component term $comp(c_1, comp(c_2, c_3))$ in $C(\Sigma)$ is formed from the composition of basic component c_1 and the composition of basic components c_2 and c_3, where $comp : C(\Sigma) \times C(\Sigma) \to C(\Sigma)$ is an operation in Σ. The inductive properties of component terms are a natural data structure for specifying BDI-agent verification plans.

Each component $c \in C(\Sigma)$ is associated with a model $m \in \mathscr{M}$ and a property $g \in \mathscr{P}$ formulated from R.

Agent Plans for Basic Components

Let c be a basic component of the component signature Σ, $m \in \mathscr{M}$ a component model and $g \in \mathscr{P}$ a property. The agent planner constructs a BDI-agent α_c with initial beliefs

$$\mathscr{B}_c = \{model(m), property(g)\} \tag{2}$$

where the predicates *model* and *property* specify agent α_c's belief of the component model m and property g. Initially, the agent has the achievement goal $!verify$ to determine the verification result $v \in \mathscr{V}$ of property g for model m. We define the context expression

$$E_c = model(M) \wedge property(G) \tag{3}$$

such that the agent holds sufficient information about its component's model and property for the verification plan constructed by the agent planner. This plan has the form

$$\begin{aligned} +verify : E_c \leftarrow & ?model(M) \wedge ?property(G) \wedge \\ & \omega(M, G, V) \wedge +result(V). \end{aligned} \tag{4}$$

Since the initial belief base (2) of α_c satisfies the plan's context E_c, the agent's goal $!verify$ triggers plan (4). The test goals $?model(M)$ and $?property(G)$ unify the model m and property g with variables M and G respectively, according to the agent's beliefs (2). The agent α_c performs the action $\omega(M, G, V)$ and invokes the oracle to compute $v = \omega(m, g, \eta)$, with the null function $\eta : \emptyset \to \emptyset$ specifying that no assumptions are required for the computation. The result v is unified with variable V and $+result(V)$ adds the value to the belief base with the predicate symbol *result*.

Agent Plans for Composite Components

We give an inductive definition of the plans for agent α_t assigned to the component term $t = f(t_1, \ldots, t_m)$ with sub-components $t_1, \ldots, t_m$ and the operation symbol $f \in \Sigma$. The agent has beliefs of its component model m and property g, and thus the initial belief base of α_t is set as $\mathscr{B}_t = \{model(m), property(g)\}$, similar to (2).

In general, the verification result $m, v \models g$ of agent α_t is dependent upon the verification values obtained by agents $\mathscr{A}_t = \{\alpha_{t_1}, \ldots, \alpha_{t_m}\}$ for the sub-terms of t. Agent α_t therefore must communicate with each agent in $\mathscr{A}_t$ and construct assumptions $\mathbf{a} : \mathscr{P} \to \mathscr{V}$ to be supplied to the oracle. To coordinate with agents $\alpha_{t_1}, \ldots, \alpha_{t_m}$, agent α_t is constructed with the plan

$$\begin{aligned} +!queryagents : true \leftarrow & \\ send(t_1, givep, A_1) \wedge \cdots \wedge & send(t_m, givep, A_m) \\ \wedge send(t_1, givev, V_1) \wedge \cdots \wedge & send(t_m, givev, V_m) \end{aligned} \tag{5}$$

which asks each sub-component agent α_{t_i} to retrieve the verification task information stored within its belief base. The properties $a_1, \ldots, a_m$ in $\mathscr{P}$ and values

$v_1, \ldots, v_m$ in $\mathscr{V}$ are unified with variables $A_1, \ldots, A_m$ and $V_1, \ldots, V_m$ respectively. They are used to construct the assumption function $\mathbf{a} : \{a_1, \ldots, a_m\} \rightarrow \{v_1, \ldots, v_m\}$ such that

$$\mathbf{a}(a_i) = v_i. \tag{6}$$

To facilitate communication with α_t, each agent α_{t_i} is constructed with the plan

$$\begin{aligned} +?givep : \ true \leftarrow & \ ?property(A) \\ & \wedge \ send(t, property_{t_i}(A)). \end{aligned} \tag{7}$$

by the agent planner, which is triggered whenever its mental state is changed to the test goal $?givep$, for $1 \leq i \leq m$.

The plan body of (7) contains a special *send* action which enables transmission of messages between agents using the Knowledge Query and Manipulation Language (KQML) [14] which expresses an agent's intention for sending a message. Agents send messages for two purposes: i) to change the receiver's beliefs, or ii) to change the receiver's goals. In the plan body of (7), agent α_{t_i}'s property belief is unified with the variable P and sent to agent α_t, who receives the belief and adds it to $\mathscr{B}_t$. Similarly, the verification value obtained from the oracle ω by α_{t_i} is sent to α_t by the plan

$$+?givev : true \leftarrow ?result(V) \wedge send(t, result_{t_i}(V)). \tag{8}$$

The goal $?result(V)$ is complete only when α_{t_i} completes its verification task and $result_{t_i}(V)$ is added to its belief base.

For the composite component represented by the component term t with sub-terms $t_1, \ldots, t_m$, the context expression (3) is extended to

$$\begin{aligned} E'_t = \ & E_t \wedge property_{t_1}(A_1) \wedge \cdots \wedge property_{t_m}(A_m) \\ & \wedge \ result_{t_1}(V_1) \wedge \cdots \wedge result_{t_m}(V_m) \end{aligned} \tag{9}$$

to ensure the values from agents $\alpha_{t_1}, \ldots, \alpha_{t_m}$ have been received. The agent planner formulates two plans so that agent α_t can achieve its verification goal $!verify$. The first plan is applicable whenever α_t has not obtained values from agents $\alpha_{t_1}, \ldots, \alpha_{t_m}$. In this context, the plan

$$+!verify : \neg E'_t \leftarrow !queryagents \ \wedge \ !verify$$

comprises the achievement subgoal $!queryagents$ which triggers plan (5) and continues to attempt the achievement $!verify$. In the context of receiving values from all sub-component agents, the plan invoked to achieve $!verify$ is

$$\begin{aligned} +verify : \ E'_t \leftarrow & ?model(M) \wedge ?property(G) \\ & \wedge \ ?property_{t_1}(A_1) \ \wedge \ \cdots \ \wedge ?property_{t_m}(A_m) \\ & \wedge \ ?result_{t_1}(V_1) \ \wedge \ \cdots \ \wedge ?result_{t_m}(V_m) \\ & \wedge \ \omega(M, G, V, A_1, \ldots, A_m, V_1, \ldots, V_m) \\ & \wedge \ +result(V). \end{aligned} \tag{10}$$

comprising additional test goals unifying $A_1, \ldots, A_m$ and $V_1, \ldots, V_m$ with properties and values from the verification tasks of agents $\alpha_{t_1}, \ldots, \alpha_{t_m}$. The properties and values construct the assumption function a defined in (6) used as input

for action $\omega(M, G, V, A_1, \ldots, A_m, V_1, \ldots, V_m)$, invoking the external oracle to compute $v = \omega(m, g, \mathbf{a})$. The variable V is unified with the value v and stored in the agent's belief base with predicate symbol $result$.

3.3 Agent Runtime Phase

Runtime environments are dynamic and changes such as the addition, removal and modification of components occur in rapid succession. In this section, we specify plans enabling an agent to sense localised changes of the components modelled by their verification task. When a component change occurs, the agent is required to adapt and evolve its beliefs and behaviour during system operation in order to maintain a verification task that correlates to the actual state of the system. Formally, agent α_t is constructed with the plan

$$+!detect : true \leftarrow sense \wedge !detect. \tag{11}$$

which executes the action *sense* to detect changes in the component $t \in C(\Sigma)$. Component change detection only occurs after α_t achieves its initial verification goal. We extend the agent's verification plan by appending the achievement goal $!detect$ to the plan bodies listed in (4) and (10).

Agent Change Perceptions

The agent action *sense* notifies the engine that the system represented algebraically by the component term S is updated according to a component change sensed in the *runtime environment*. Mathematically, a component change is modelled as a component term $t' \in C(\Sigma)$ with an associated model $m' \in \mathscr{M}$ and property $g' \in \mathscr{P}$. When a component change occurs, the engine first updates S and associated model and property. It then notifies agent α_t of the new model m' and property g', and finally adds a *change perception* to the belief base of agent α_t corresponding to addition, removal or modification changes.

Modification Component Change

Change perceptions are handled by agents in the following cases. When component t_i of the component $t \equiv f(t_1, \ldots, t_m)$ for $1 \leq i \leq m$ is modified, a change perception $modified$ is issued by the engine to agent α_{t_i}. In this case, the engine first updates the component model and property to m'_{t_i} and g'_{t_i} respectively. The engine updates the belief base $\mathscr{B}_{t_i}$ of the agent, replacing its model and property beliefs with information of the new verification task $model(m'_{t_i})$ and $property(g'_{t_i})$, respectively. The engine's change perception triggers the belief addition event $+modified$ in the agent and is handled by the plan

$$+modified : true \leftarrow !verify \wedge !notify.$$

supplied by the agent planner. The agent proceeds to call the oracle to verify its modified task, providing the agent with a new belief $result(V)$. Next, the agent achieves the goal $!notify$ to notify agent α_t that re-verification is complete. The plan

$$+!notify : true \leftarrow send(t, !reverify_{t_i}). \tag{12}$$

communicates the $!reverify_{t_i}$ achievement goal to agent α_t. Agent α_t attempts to achieve this goal using the plan

$$\begin{aligned} +!reverify_{t_i} : true \leftarrow & - result_{t_i} \wedge -property_{t_i} \\ & \wedge send(t_i, givep, A_i) \\ & \wedge send(t_i, givev, V_i) \wedge !verify \wedge !notify. \end{aligned} \tag{13}$$

which obtains the modified property a'_i and updated verification result v'_i from α_{t_i}. The $!verify$ goal uses the updated assumptions function $\mathbf{a}' : \mathscr{P} \rightarrow \mathscr{V}$ to invoke the oracle where

$$\mathbf{a}'(a_j) = \begin{cases} v'_i & \text{if } a_j = a_i, \\ \mathbf{a}(a_j) & \text{otherwise,} \end{cases}$$

and notifies agents that re-verification has occurred.

Addition Component Change

In the general case, the component term $t = f(t_1, \ldots, t_m)$ has m sub-components. The addition of the new component t_{m+1} to t is specified by the component term $t' = f(t_1, \ldots, t_m, t_{m+1})$. Mathematically, we define the substitution operation

$$S' = S[f(t_1, \ldots, t_m)/f(t_1, \ldots, t_m, t_{m+1})] \tag{14}$$

to replace instances of t with t' in the component term representation S to obtain S'. When the system representation is updated, the engine adds the change perception $add(t_{m+1})$ to agent α_t's belief base, triggering a belief addition event that is handled accordingly by the plan

$$\begin{aligned} +add(T) : true \leftarrow & create(T) \ \wedge \ replan \\ & \wedge !verify \ \wedge \ !notify. \end{aligned}$$

The action $create(T)$ invokes the agent planner to construct a new agent $\alpha_{t_{m+1}}$ and assign it the verification task w associated with the new component t_{m+1}. Once constructed, the agent completes its initial achievement goal $!verify$. The action $replan$ described in the next section is used by α_t to take into account the added component. Once α_t obtains new plans it re-verifies to take into account the new component. The goal $!notify$ ensures all relevant agents are notified.

Removal Component Change

A component t_i is removed from $t = f(t_1, \ldots, t_i, \ldots, t_m)$, resulting in a new component term $t' = f(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m)$. The term substitution operation (14) is applied to S such that $S' = S[f(t_1, \ldots, t_i, \ldots, t_m)/f(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m)]$ replaces instances of t with t' in S, forming the component term S'. When the system representation is updated, the engine adds the change perception $remove_{t_i}$ to agent α_t's belief base, triggering a belief addition event handled by the plan

$$\begin{aligned} +remove_{t_i} : true \leftarrow & - property_{t_i} \wedge -result_{t_i} \wedge \ stop_{t_i} \\ & \wedge \ replan \ \wedge \ !verify \ \wedge \ !notify. \end{aligned}$$

which removes agent α_t's beliefs regarding the verification result and property obtained from agent α_{t_i}. The action $stop_{t_i}$ terminates and removes agent α_{t_i} from the AVE. The agent plan *replan* requests new plans that enable α_t to reverify their verification task and notify all relevant agents.

Agent Evolution

Agent action *replan* requests advice from the planner after a change. The planner modifies the agent's plan library as follows. Given the component term t' obtained by adding or removing the sub-component term t_i from $t = f(t_1, \ldots, t_i, \ldots, t_m)$, the agent plan is modified according to

- a new *query* goal (5) adding or removing actions $send(t_i, givep, A_i)$ and $send(t_i, givev, V_i)$ requesting agent α_{t_i}'s property and verification value. This corresponds to constructing the assumption function in (6).
- a context rule (9), adding or removing $property_{t_i}(A_i)$ and $result_{t_i}(V_i)$ as belief requirements for verification,
- a verification plan body (10), adding or removing test goals $?property_{t_i}(A_i)$ and $?test_{t_i}(V_i)$. The parameter listing in the oracle invocation is accordingly updated to add or remove parameters P_i and V_i.
- a re-verification plan (13), adding and removing plans corresponding to sub-components dependencies.

When the agent receives the new plans, the agent reinitialises and attempts to achieve their initial verification goal.

Re-verification Policies

Plans (12) and (13) perform re-verification that essentially follow recently introduced incremental verification techniques [15, 13] based on component dependencies defined by the system's architecture. By using a BDI-agent approach, these techniques are extendable to generalised *verification policies* comprising programmable agent behaviours to perform remedial actions taken in response to component change. Agent behaviour can express policies to send conditional notifications only when their component becomes non-compliant to requirements after a change, perform verification steps involving other oracles, or notify other agents specified by the application domain requirements.

4 A Compositional Approach to SMT

We instantiate AVE with an existing SMT-based technique. The following concepts are defined in [15] and reproduced for completeness.

Component Models

To integrate an SMT verification service with the engine we define $\mathscr{M} = \mathbb{P}(Z)^+$, such that each component term t is modelled as a finite subset Z_t of variables from the set Z inductively on the structure of component terms:

- $Z_{c_i} \in \mathbb{P}(Z)$, for each basic component term c_i such that models are pairwise disjoint $Z_{c_i} \cap Z_{c_j} = \emptyset$, for $i \neq j$.
- $Z_{f(t_1,\ldots,t_m)} = \cup_{i=1}^{m} Z_{t_i}$ for each operation $f \in \Sigma$ and component terms $t_1, \ldots, t_m$.

Component Properties

Let $\mathscr{P} = F(\Gamma, Z)$ such that component requirements are quantifier-free first order formulae over the signature Γ, defined inductively by the rules

$$\begin{aligned}\phi ::= & t_1 = t_2 \mid r_1(t_1, \ldots, t_{n_1}) \mid \cdots \mid r_m(t_1, \ldots, t_{n_m}) \mid \\ & \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2\end{aligned}$$

where t_1 and t_2 are terms, and ϕ_1 and ϕ_2 are formulae.

We often display the variables that appear in a formula by writing $\phi(\mathbf{z})$ to mean that the formula ϕ has at least one instance of the variables in the tuple $\mathbf{z} = (z_1, \ldots, z_p)$.

SMT Resolution

Satisfiability modulo theories represent a class of theoretical techniques and practical tools that determine the satisfiability of a formula expressed in terms of tests and operations in a signature Γ. An SMT technique $smt : F(\Gamma, Z) \to [Z \to \mathscr{V}]$ is applied to the formula $\phi(\mathbf{z})$ in order to compute and return an assignment $\mathbf{v} : Z \to \mathscr{V}$ of values from the set $\mathscr{V}$ to the variables in $\mathbf{z}$, such that the formula is satisfied. In symbols, we write $\mathbf{v} \models \phi(\mathbf{z}) \iff [\![\phi(\mathbf{z})]\!](\mathbf{v}) = true$ where $\mathbf{v} = smt(\phi(\mathbf{z}))$. If no such $\mathbf{v}$ exists then the unsatisfiable assignment $\mathbf{u} : Z \to \mathscr{V}$ is returned. We note the empty formula ϵ is valid (i.e. satisfied by all assignment mappings).

The requirements function $R : C(\Sigma) \to \mathscr{P}$ supplied to the agent verification engine associates each component term t with a logical formula ϕ_t such that $var(\phi_t) \subseteq Z_t$ where $var(\phi_t)$ is the set of variables in ϕ_t and Z_t is the component model of t. We use the inductive properties of component terms to compute the conjunction $\phi = \phi_{t_1} \wedge \cdots \wedge \phi_{t_m}$ of formulae associated to the components $t_1, \ldots, t_m$ of S. By resolving ϕ using smt, we get an assignment $\mathbf{a} : \mathscr{P} \to \mathscr{V}$ of values from $\mathscr{V}$ to variables in ϕ such that system requirements R are satisfied. We also define the function $mono : C(\Sigma) \to F(\Gamma, Z)$ such that $mono(t)$ is the monolithic formula of t, by induction over the structure of component terms.

Let t be a component term in $C(\Sigma)$.
Base case: for $t \equiv c_i$ the basic component c_i, we have $mono(c_i) = \phi_{c_i}$.
Inductive step: for $t \equiv f(t_1, \ldots, t_m)$ with sub-components $t_1, \ldots, t_m$ and operation symbol $f \in \Sigma$, we have $mono(t) = \phi_{f(t_1,\ldots,t_m)} \wedge (\wedge_{i=1}^{m} mono(\phi_{t_i}))$.

Compositional SMT Resolution

As the component-based system S typically contains a large number of components and requirements, resolving a monolithic formula $mono(S)$ involves a huge constraint problem and is generally unfeasible for a single agent to complete during runtime. Instead, we describe a compositional approach based on the results of [15, 16] which constructs smaller resolution steps to be carried out independently. The solutions obtained in each step are combined to form a solution for the monolithic formula.

The resolution of the conjunction $\phi_1(\mathbf{y}) \wedge \phi_2(\mathbf{z})$ of independent formulae (no shared variables) may be decomposed into two smaller steps $a_1 = smt(\phi_1(\mathbf{y}))$ and $a_2 = smt(\phi_2(\mathbf{z}))$, combined using the operation $\oplus : [\mathscr{P} \to \mathscr{V}]^2 \to [\mathscr{P} \to \mathscr{V}]$ such that $(a_1 \oplus a_2)(z) = a_1(z)$ if $z \in \mathbf{z}$ and $a_2(z)$ otherwise. We can

prove that $a_1 \oplus a_2 \models \phi_1(\mathbf{y}) \wedge \phi_2(\mathbf{z})$ (cf. Lemma 4.1 [15]). We extend this observation to $n > 1$ independent logical formulae $\phi_1, \ldots, \phi_n$ and construct an assignment $v \models \wedge_{i=1}^{n} \phi_i$ through a sequence of independent resolution steps $v_1 \models \phi_1, \ldots, v_n \models \phi_n$. (cf. Theorem 4.1 [15]).

The SMT Oracle

Compositional SMT resolution serves as the basis for instantiating the assume-guarantee oracle ω specified by (1) to perform compositional SMT resolution. The oracle accepts as input a *verification task* comprising

- mapping $\mathbf{a} : \{\phi_1, \ldots, \phi_n\} \rightarrow \{v_1, \ldots, v_n\}$ of independent logical formulae in $F(\Gamma, Z)$ to values in $\mathscr{V}$ representing assumptions such $v_i \models \phi_i$ for $\phi_i \in dom(\mathbf{a})$ and $v_i \in range(\mathbf{a})$,
- $g \in F(\Gamma, Z)$, independent of each formula in $dom(\mathbf{a})$,
- model $m = var(g) \cup (\cup_{i \in \{1,\ldots,n\}} var(a_i))$, the union of the variable sets for g and $a_1, \ldots, a_n$.

We model the oracle as the total function $\omega : W \rightarrow \mathscr{V}$ as

$$\omega(m, g, \mathbf{a}) = smt(g) \oplus (\oplus_{i=1}^{n} v_i) \tag{15}$$

that performs *smt* resolution to compute an assignment of values in $\mathscr{V}$ to the model variables in m that satisfies the conjunction $g \wedge a_1 \wedge \cdots \wedge a_n$ of logical formulae. Thus, we conclude $\omega(m, g, \mathbf{a}) \models g \wedge (\wedge_{i=1}^{n} a_i)$.

5 Case Study

We introduce an industrial control system to illustrate our agent-based verification engine. Figure 2 depicts an item sorting system (ISS), typically used to sort high volume item streams within systems such as airport baggage handling and food packaging. Items are placed in an initial position by a robotic arm. Horizontal and vertical pushers use sensors to detect items and then direct them towards user-specified end-points. Additional robotic arms remove items from the end-points. The system software is written using the IEC 61499 [17] language, which enables component-based design where components contain finitely nested networks of (sub) components. Figure 3 shows the architecture of the ISS. The system **S** comprises components **vert**, **horiz** and **robot**, representing the control programs for the vertical and horizontal pushers, and the robotic arms, respectively. Each component has three sub-components labelled from **A** to **I**.

Components of IEC 61499 programs execute on *devices* such as programmable logic controllers (PLCs). Components deployed onto the same PLC communicate using faster local mechanisms such as shared buffers. Components on different PLCs communicate using networks like Ethernet. For the ISS, which handles high item volumes, system speed can be significantly reduced if critical path sub-components like communicating pushers execute on different PLCs. Also, the ISS can be made more dependable by deploying multiple instances of sub-components $\mathbf{A} - \mathbf{I}$ across available PLCs, numbered 1 to p ($p \geq 2$), according to the following high-level requirements R:

R1 each component has 1 to 10 instances deployed

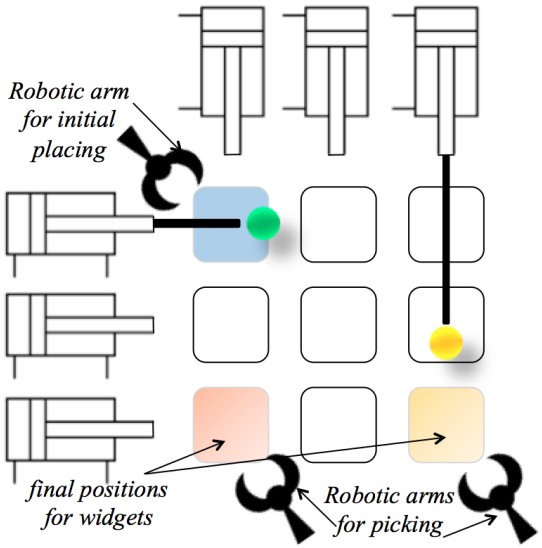

Figure 2: An item sorting system

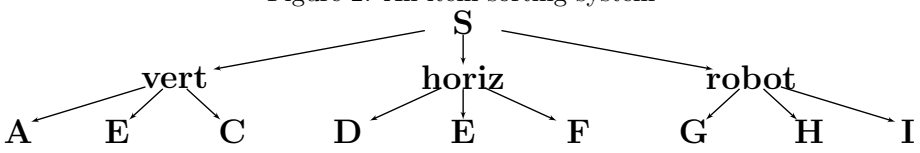

Figure 3: Architecture of the item sorting system

R2 $\mathbf{A} - \mathbf{C}$ have at least 4 instances deployed
R3 **G** has exactly 3 instances on PLC 2
R4 **H** must not be deployed on PLC 1
R5 all instances of **D**, **E** and **F** are identically distributed

The *deployment configuration problem* determines a distribution of component instances of the ISS **S** across p PLCs to satisfy the high-level requirements R.

System Specification in AVE: We define a component signature Σ and express an algebraic specification of the system architecture depicted in Figure 3. Let the set of component terms be denoted as C. We set the basic components of the system $\mathbf{A}, \ldots, \mathbf{I}$ and define an operation symbol $c : C \times C \to C$. For the ISS, we have the component term

$$\mathbf{S} \equiv c(\mathbf{vert}, c(\mathbf{horiz}, \mathbf{robot})) \tag{16}$$

where, $\mathbf{vert} \equiv c(\mathbf{A}, c(\mathbf{B}, \mathbf{C}))$, $\mathbf{horiz} \equiv c(\mathbf{D}, c(\mathbf{E}, \mathbf{F}))$ and $\mathbf{robot} \equiv c(\mathbf{G}, c(\mathbf{H}, \mathbf{I}))$.

Component models: To apply the compositional SMT resolution technique described in Section 4 to the deployment configuration problem, we model subcomponents $\mathbf{A}, \ldots, \mathbf{I}$ by finite sets $Z_{\mathbf{A}}, \ldots, Z_{\mathbf{I}}$ of variables. E.g., **A** is modelled by the set $Z_{\mathbf{A}} = \{\mathbf{Az_1}, \ldots, \mathbf{Az_p}\}$ of integer-typed variables from the set Z such that the assignment $\mathbf{Az_i} = n$ models the deployment of n instances of component **A** onto the i^{th} PLC. By definition, we have the model $Z_{\mathbf{S}} = Z_{\mathbf{vert}} \cup Z_{\mathbf{horiz}} \cup Z_{\mathbf{robot}}$ associated with the component term **S**, where $Z_{\mathbf{vert}} = Z_{\mathbf{A}} \cup Z_{\mathbf{B}} \cup Z_{\mathbf{C}}$,

$Z_{\mathbf{horiz}} = Z_{\mathbf{D}} \cup Z_{\mathbf{E}} \cup Z_{\mathbf{F}}$ and $Z_{\mathbf{robot}} = Z_{\mathbf{G}} \cup Z_{\mathbf{H}} \cup Z_{\mathbf{I}}$.

Requirements Specification We define a mapping $\phi : C(\Sigma) \rightarrow F(\Gamma, Z)$ by formalising requirements **R1** - **R5** of the ISS architecture as logical formulae over integer-type variables to be associated with components of **S**. Let Γ be a signature comprising sorts for integers and Boolean, symbols for the standard arithmetic operations, equality and inequality predicates. Initially, we set $\phi_t = \epsilon$, that signifies the case where component t is not constrained. Requirement **R1** sets the range of deployed instances for each sub-component of **S** between 1 and 10, with non-negative number of instances over any PLC. For sub-component **A**, we express this requirement as the formula

$$\phi^1_{\mathbf{A}} := \bigwedge_{i=1}^{p} \mathbf{A}_{z_i} \geq 0 \wedge \sum_{i=1}^{p} \mathbf{A}_{z_i} \geq 1 \wedge \sum_{i=1}^{p} \mathbf{A}_{z_i} \leq 10. \tag{17}$$

Similarly, all other sub-components $\mathbf{B} - \mathbf{I}$ are assigned formulae $\phi^1_{\mathbf{B}} - \phi^1_{\mathbf{I}}$ respectively. Requirement **R2** further constrains the deployment, and is expressed as $\phi^2_{\mathbf{A}} := \sum_{i=1}^{p} \mathbf{A}_{z_i} \geq 4$. We add this formula to the existing formula for **R1** by assigning $\phi_{\mathbf{A}} := \phi^1_{\mathbf{A}} \wedge \phi^2_{\mathbf{A}}$. We assign simular formulae $\phi_{\mathbf{B}} := \phi^1_{\mathbf{B}} \wedge \phi^2_{\mathbf{B}}$ and $\phi_{\mathbf{C}} := \phi^1_{\mathbf{C}} \wedge \phi^2_{\mathbf{C}}$ for sub-components **B** and **C**. **R3** is formalised as $\phi^2_{\mathbf{G}} := (\mathbf{Gz_2} = 4)$ and we construct $\phi_{\mathbf{G}} := \phi^1_{\mathbf{G}} \wedge \phi^2_{\mathbf{G}}$, using the formula $\phi^1_{\mathbf{G}}$ formalising **R1** for **G**. For **R4** we specify $\phi^2_{\mathbf{H}} := (\mathbf{Hz_1} = 0)$ and add this formula to the existing requirements by assigning $\phi_{\mathbf{H}} := \phi^1_{\mathbf{H}} \wedge \phi^2_{\mathbf{H}}$. Lastly, we formalise Requirement **R5** which states that the sub-components of $\mathbf{D}, \mathbf{E}$ and **F** must have identical distributions of instances across the PLCs. We define the formula

$$\phi_{\mathbf{vert}} := \bigwedge_{i=1}^{p} (\mathbf{Dz_i} = \mathbf{Ez_i}) \wedge (\mathbf{Fz_i} = \mathbf{Ez_i}) \tag{18}$$

where the model for **vert** comprises variables from its sub-component models $Z_{\mathbf{D}}$, $Z_{\mathbf{E}}$, and $Z_{\mathbf{F}}$.

6 Implementation and Simulation

We developed a generic multi-agent simulator of the agent verification engine as a open-source Java application. The engine's implementation is based on Jason AgentSpeak [10]. Jason has a theoretical basis which is amenable to our formal approach for system adaptation and has built-in, extensible support for multi-agent system distribution over networks. We used Jason's standard architecture for agent perceptions, inter-communication and actions. The AVE's implementation comprises the following core classes

- `RunTimeEnvironment<M,P,V>` simulates perceptible component changes within the system's runtime environment and executes agent actions (`M` for models, `P` for properties and `V` for verification results)
- `Engine<M,P,V>` maintains the system representation and compositional verification technique
- `AgentPlanner<M,P,V>` constructs agents and provides new plans after changes occur.
- `CompVerify`, a compositional verification paradigm providing abstract methods `add`, `modify` and `remove` for updating the system's algebraic specification.

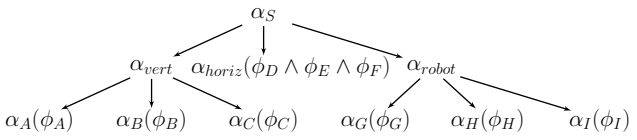

Figure 4: ISS Agent Tree

Table 1: An assignment mapping satisfying all ISS requirement in R for 2 PLCs.

	α_A	α_B	α_C	$\alpha_{\mathbf{vert}}$			α_G	α_H	α_I
$v_{\mathbf{S}}$	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
z_1	1	4	2	1	1	1	0	0	1
z_2	3	0	2	1	1	1	3	1	0

Agent Verification Engine for the ISS: We instantiated AVE for the deployment configuration problem for the ISS as follows. First, `Z3CompVerify`, a concrete class utilising the compositional SMT approach described in Section 4 was developed. The oracle ω defined by (15) is implemented using the Microsoft satisfiability modulo theories solver Z3 [18]. AVE was also provided with $\mathbf{S}$, the architecture (16) of the ISS, as well as a mapping $\phi : C(\Sigma) \rightarrow F(\Gamma, Z)$ that assigns logical formulae to components of the system.

When the engine receives these inputs, the agent planner constructs the necessary agents that co-operate to solve the deployment configuration problem. Figure 4 depicts a tree comprising ten agents in the set $\mathscr{A}$ constructed by the agent planner as described in Section 3. Each node is labeled with the name of the agent corresponding to a component in the ISS. The logical formulae to be resolved by the agent using the oracle are written in parenthesis. The formula $\phi_{\mathbf{vert}}$ defined in (18) shares variables with its sub-component formulae $\phi_{\mathbf{D}}$, $\phi_{\mathbf{E}}$ and $\phi_{\mathbf{F}}$. None of these formulae can be resolved independently thus the engine assigns the task of resolving $\phi_{\mathbf{vert}} \wedge \phi_{\mathbf{D}} \wedge \phi_{\mathbf{E}} \wedge \phi_{\mathbf{F}}$ to agent $\alpha_{\mathbf{vert}}$. The edges of the tree denote communication links between agents in which queries, properties and values are transmitted according to plans (5), (7) and (8) respectively.

Once the engine has been initialised, agents in $\mathscr{A}$ attempt to achieve their verification goal. We consider this process for agent $\alpha_{\mathbf{A}}$ as follows. The agent has initial beliefs $model(Z_A)$ and $property(\phi_{\mathbf{A}})$. It then attempts to achieve goal $!verify$ using plan (10) by invoking $\omega(Z_{\mathbf{A}}, \phi_{\mathbf{A}}, \eta)$ to obtain assignment $v_{\mathbf{A}} = \{\mathbf{Az_1} \rightarrow 1, \mathbf{Az_1} \rightarrow 3\}$, storing the belief $result(v_{\mathbf{A}})$. It also attempts the achievement goal $!detect$ defined in (11) to sense changes in component $\mathbf{A}$ during runtime.

The configuration problem is resolved when the result belief is stored by $\alpha_{\mathbf{S}}$. $\alpha_{\mathbf{S}}$ carries out the following steps for this purpose. $\alpha_{\mathbf{S}}$ first queries the agents $\alpha_{\mathbf{vert}}$, $\alpha_{\mathbf{horiz}}$ and $\alpha_{\mathbf{robot}}$ and forms the assumption function $a : \mathscr{P} \rightarrow \mathscr{V}$ such that $a(\phi_{\mathbf{vert}}) = v_{\mathbf{vert}}$, $a(\phi_{\mathbf{horiz}}) = v_{\mathbf{horiz}}$, and $a(\phi_{\mathbf{robot}}) = v_{\mathbf{robot}}$. $\alpha_{\mathbf{S}}$ then invokes the oracle $v_{\mathbf{S}} = \omega(Z_{\mathbf{S}}, \epsilon, a)$, where $Z_{\mathbf{S}}$ is the component model of $\mathbf{S}$, ϵ is the empty formula. Finally, $\alpha_{\mathbf{S}}$ stores $+result(v_{\mathbf{S}})$ in its belief base.

Table 1 depicts the value $v_{\mathbf{S}}$ obtained by the SMT process carried out by the oracle. The agent's belief represents one possible assignment mapping to satisfy all requirements in R.

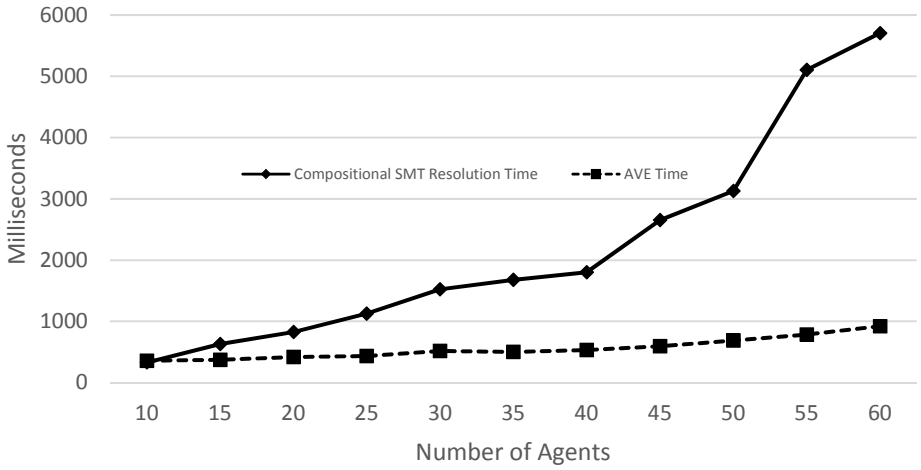

Figure 5: Configuration using Compositional SMT resolution and AVE

Experimental Results

We evaluated the performance of AVE by running a series of experiments starting with the ISS architecture shown in Figure 3 and subsequently increasing the system size by five components at each step, corresponding to an increase of 5 agents and 50 variables in the Z3 specification of the system requirements (assuming 10 available PLCs). The experiments hence considered system sizes containing between 10 and 60 agents. A standard PC running Windows 7 Enterprise 64-bits with an Intel i7 2.1GHz Processor and 16 GB RAM was used for the tests. We measured the time for $\alpha_{\mathbf{S}}$ to acquire the belief $result(v_{\mathbf{S}})$ and compared it to the cumulative time across all oracle invocations, corresponding to the time needed to complete minimal number of steps to resolve the deployment configuration problem using standard compositional SMT resolution. The results of this experiment are presented in Figure 5. All runs completed under 7 seconds. For small system sizes, the difference in speed of the two approaches is negligible. As the system grows with additional components, AVE benefits from concurrent oracle invocations by the agents.

We also carried out the following experiments to evaluate the performance of our implementation for *re-configuration* after a system change is perceived by an agent.

1. Adding a Robotic Arm: The ISS requires an additional robotic arm to pick up items from a new end-point and the runtime environment simulates the addition of a new component $\mathbf{R}$ to the system. The component is modelled by the variable set $Z_{\mathbf{R}}$ and its requirements are formalised by the standard logical formula $\phi_{\mathbf{R}} = \phi^1_{\mathbf{R}}$ as defined in (17). The agent verification engine performs the following steps. First, a substitution on the component term $\mathbf{S}$ is performed, obtaining $\mathbf{S}' \equiv \mathbf{S}[\mathbf{robot}/c(c(\mathbf{G}, c(\mathbf{H}, \mathbf{I})), \mathbf{R})]$ and $add(\mathbf{R})$ is added to $\alpha_{\mathbf{robot}}$ belief base. Next, agent $\alpha_{\mathbf{robot}}$ issues the action to $create(\mathbf{R})$. The agent planner then submits new plans to $\alpha_{\mathbf{robot}}$, who invokes the oracle with assumptions from $\alpha_{\mathbf{G}}$ to $\alpha_{\mathbf{I}}$ and $\alpha_{\mathbf{R}}$. Agent $\alpha_{\mathbf{R}}$ obtains a new belief of the result. Finally, $\alpha_{\mathbf{robot}}$ notifies $\alpha_{\mathbf{S}}$, resulting in a new result belief for $\alpha_{\mathbf{S}}$ that includes the new verification result.

Figure 6 shows AVE consistently took about 37 milliseconds to re-configure

the system, ranging in size from 10 to 60 agents. In contrast, global reconfiguration of the entire system took increasingly longer as system size increased.
2. Modifying Requirements: We considered updated requirements to change the deployment of the pushers over available PLCs. Such updates may happen often for the ISS, in order to react to changing item volumes and types. As Figure 6 shows, the re-configuration effort grew from about 67 milliseconds to 110 milliseconds for systems sized from 10 to 60 agents. In contrast, a complete re-configuration can take as much as nine times longer to reverify, requiring 1000 milliseconds to completely reverify the largest system containing 60 agents. Experiments to compare re-verification time after an agent removal due to component failure showed that on average it takes 58 ms for agent $\alpha_{\mathbf{S}}$ to obtain an updated result belief, whereas global reconfiguration takes 897 ms on average, for systems containing 10 to 60 agents.

7 Related Work

There have been many approaches to solve the system reconfiguration problem that we used to demonstrate the agent verification engine. Various formal notations [19, 20, 21, 22, 23] specify and verify component-based systems whose architectures can evolve at run-time with the addition or removal of components. We address a similar problem in this paper but by contrast, our work delegates verification tasks to agents rather than handling them in a centralised way. In [24], two configuration protocols for deploying a cloud application over multiple virtual machines are proposed and verified using formal methods. In contrast to our work, the protocols proposed in this article do not take into account component failures and subsequent re-configuration actions. In [25], a reconfiguration protocol is proposed that can handle any number of failures during a reconfiguration. This protocol expects reconfiguration decisions to be made before-hand, unlike our approach which uses agents to make reconfiguration decisions based on changes in the system architecture, requirements, or device availability. Another approach that incorporates agent technology is Lira [26], a light-weight, agent-based reconfiguration engine. Each component in a given distributed software system has a unique agent associated with it. An agent handles reconfiguration requests for its associated component. RECoMa [27] is a configuration manager can help find appropriate computer platforms to deploy software agents of a multi-agent system. All available devices on which available components can be deployed are treated as equal, each capable of deploying the components allocated to it. However, our framework can be extended to heterogeneous platforms by the use of requirements that specify constraints on the device's deployment.

8 Concluding Remarks

This paper presents a generic Agent Verification Engine in which agents observe components and determine their compliance to system requirements using a supplied compositional verification technique. Agents communicate verification results such that validation of localised components infer validation of the entire system. A key benefit of the approach is the inherently distributed nature of

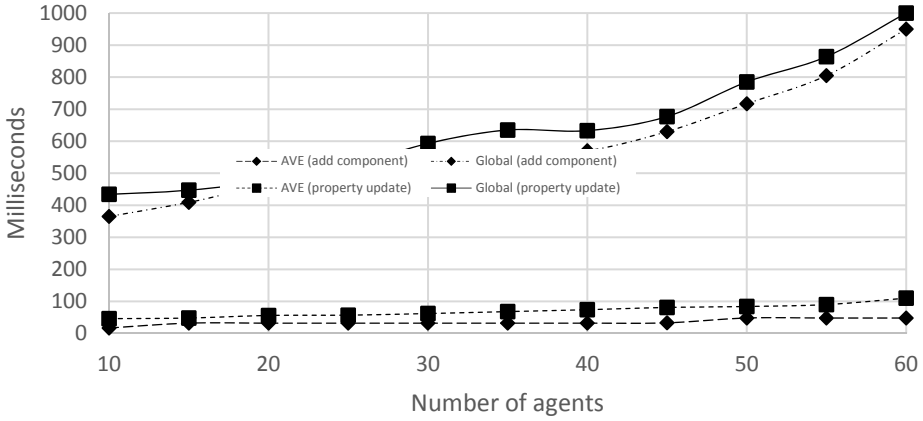

Figure 6: Running time of AVE versus full compositional verification

the agents constructed by the engine, providing a decentralised method to apply compositional verification to large-scale component-based systems.

Future work includes extending the theoretical underpinnings of our approach. We defined the engine as a translation from an algebraic specification of the system architecture to agent behaviours. BDI-agents were chosen to express re-verification behaviours since they have a rich logical framework [28] that is required to prove the correctness of the agents constructed by the engine. Secondly, the engine can be extended to include agent interpretation of verification results, forming instructions to be provided as input for actuators to affect the environment in a way that component compliance is restored after a change. Lastly, there is no end to the kinds of verification techniques to instantiate the engine and solve domain specific problems from areas such as industrial control systems or cloud computing technologies.

References

[1] I. Sommerville *et al.*, "Large-scale complex IT systems," *COMMUN ACM*, vol. 55, no. 7, pp. 71–77, 2012.

[2] S. Berezin, S. V. A. Campos, and E. M. Clarke, "Compositional reasoning in model checking," in *COMPOS*, 1997, pp. 81–102.

[3] S. Bensalem *et al.*, "Compositional verification for component-based systems and application," *IET Software*, vol. 4, no. 3, pp. 181–193, 2010.

[4] F.-S. Hsieh and J.-B. Lin, "A self-adaptation scheme for workflow management in multi-agent systems," *J Intell Manuf*, pp. 1–18, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10845-013-0818-y

[5] D. Talia, "Clouds meet agents: Toward intelligent cloud services," *IEEE Internet Computing*, vol. 16, no. 2, pp. 78–81, 2012.

[6] M. Metzger and G. Polakow, "A survey on applications of agent technology in industrial process control," *IEEE Trans. Ind. Informat.*, vol. 7, no. 4, pp. 570–581, 2011.

[7] W. Lepuschitz *et al.*, "Toward self-reconfiguration of manufacturing systems using automation agents," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 41, no. 1, pp. 52–69, 2011.

[8] A. S. Rao, "Agentspeak (l): BDI agents speak out in a logical computable language," in *Agents Breaking Away*. Springer, 1996, pp. 42–55.

[9] M. Fisher, L. Dennis, and M. Webster, "Verifying autonomous systems," *Communications of the ACM*, vol. 56, no. 9, pp. 84–93, Sep. 2013. [Online]. Available: http://doi.acm.org/10.1145/2494558

[10] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[11] K. Meinke and J. V. Tucker, "Universal algebra," in *Handbook of Logic in Computer Science*. Oxford University Press, 1992, pp. 189–368.

[12] O. Grumberg and D. E. Long, "Model checking and modular verification," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 843–871, 1994.

[13] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *CBSE*. ACM, 2013, pp. 33–42.

[14] Y. Labrou and T. Finin, "Semantics and conversations for an agent communication language," in *Readings in Agents*. Morgan Kaufmann, 1998, pp. 235–242.

[15] K. Johnson and R. Calinescu, "Efficient re-resolution of SMT specifications for evolving software architectures," in *QoSA*. ACM, 2014, pp. 93–102.

[16] R. Sinha, K. Johnson, and R. Calinsescu, "A scalable approach for reconfiguring evolving industrial control systems," in *ETFA*, 2014.

[17] R. Lewis, *Modelling distributed control systems using IEC 61499: Applying function blocks to distributed systems*. IET, 2001, no. 59.

[18] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*. Springer, 2008, pp. 337–340.

[19] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *FASE*, 1998, pp. 21–37.

[20] M. Wermelinger, A. Lopes, and J. L. Fiadeiro, "A graph based architectural (re)configuration language," in *ACM SIGSOFT SEN*, 2001, pp. 21–32.

[21] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *ICSE*, 2006, pp. 371–380.

[22] S. S. Kulkarni and K. N. Biyani, "Correctness of component-based adaptation," in *CBSE*, vol. 3054. Springer, 2004, pp. 48–58.

[23] A. Cansado *et al.*, "A formal framework for structural reconfiguration of components under behavioural adaptation," *Electr. Notes Theor. Comput. Sci.*, vol. 263, pp. 95–110, 2010.

[24] G. Salaün *et al.*, "An experience report on the verification of autonomic protocols in the cloud," *Innov Syst Softw Eng*, vol. 9, no. 2, pp. 105–117, 2013.

[25] F. Boyer, O. Gruber, and D. Pous, "Robust reconfigurations of component assemblies," in *ICSE*. IEEE Press, 2013, pp. 13–22.

[26] M. Castaldi *et al.*, "A lightweight infrastructure for reconfiguring applications," in *ICSE*, 2003, pp. 231–244.

[27] J. A. Giampapa, O. H. Juarez-Espinosa, and K. P. Sycara, "Configuration management for multi-agent systems," in *AAMAS*. ACM, 2001, pp. 230–231.

[28] A. S. Rao and M. P. Georgeff, "Decision procedures for BDI logics," *J Logic Comput*, vol. 8, no. 3, pp. 293–343, 1998.