Detecting Job Interference in Large Distributed Multi-Agent Systems — A Formal Approach

Wenjie Lin*, Michael McGrath*, Ingy Ramzy*, Ten-Hwang Lai* and David Lee†
*The Ohio State University
linw@cse.ohio-state.edu, mcgrath.57@buckeyemail.osu.edu, {youssef, lai}@cse.ohio-state.edu
†HP Labs
david.lee10@hp.com

***Abstract*—This work is on formal modeling, analysis and detection of job interference in large distributed multi-agent systems. Such an analysis usually requires an examination of all the global system states—often impossible due to the well-known state space explosion. We obtain a sufficient condition so that job interference can be detected by observations of individual system component without the knowledge of global system states.**

Given that the job interference can be detected locally, we propose a guided random walk algorithm for detecting interference. We apply it to Kansei, a large and distributed wireless sensor network system with multi-agents. Ten job interference traces are identified; they have not been detected before by manual analysis and system operations. We further diagnose the detected interference for a correction of system design.

I. Introduction

In a general distributed multi-agent system, agents operate together to provide functions for various jobs. Independent jobs may coexist and operate asynchronously in a system with their agents potentially residing on the same node. For example, a wireless sensor network testbed is a multi-agent system where multiple users' experiments (*i.e.* jobs) can be scheduled on a single node—consisting of one or more sensors. As the logic of each individual job are designed independently—lack of knowledge of other jobs while jobs may execute simultaneously and asynchronously, interference among the jobs may arise when they are hosted together.

Interference between independent jobs often lead to severe anomalies in multi-agent systems[1]. In a large-scale wireless sensor network testbed, a management job that controls system time can cause experiment data to be overwritten when it interferes with experiment jobs. This is observed in Kansei system when daylight saving time starts [1]. In smart building systems, false alarms are found, which are generated by interference between the security job and the climate control job: the former raises an alarm because of the window movement triggered by the latter [2].

Detecting job interference is thus desired in multi-agent systems. There are two general approaches: on-line detection and off-line detection. Off-line detection is preferred when interference needs to be detected before jobs execute, or when resources are too limited to afford the on-line detection (*e.g.* in real-time systems or in energy-limited wireless systems). Even in the systems in which on-line monitors are deployed, off-line checking in advance can reveal system design flaws, thus simplifies the schemes of on-line detection and resolution. However, it is far from easy to detect job interference in large distributed multi-agent systems off line. The traditional model checking approaches—seeking interference by generating a global reachability graph from the system model—have a key challenge: state explosion. Because the system state space (in worst case the Cartesian product of states in each node) exponentially expands as the number of nodes increases, automated model checking on large distributed multi-agent systems can easily fail before any concrete results are obtained. Our primitive attempt on running model checker SPIN [3] against the Kansei model quickly runs out of memory. Although general methods and heuristics—such as partial order reduction, abstraction, symmetry [4], local variable reset [5], predicate simplification [6], and minimized automaton encoding [3]—can ease the state explosion due to the number of nodes, they usually come with the cost of completeness.

A question is raised accordingly: for job interference detection in distributed multi-agent systems, is there a way to avoid state explosion generated by the large number of nodes?

In this paper, we focus on off-line interference detection among independent jobs in large distributed multi-agent systems. We aim at a practical solution that scales when the number of nodes grows. The following are our three contributions.

First, for the off-line interference detection in distributed multi-agent systems, we discover that, if the model is fine-grained and the control & management components are well-extended, local detection node-by-node is equivalent to global detection with *soundness* (*i.e.* no false positive) and *completeness* (*i.e.* no false negative).

Second, we propose a guided random walk algorithm to obtain a job's all local behaviors and thus detect local job interference. Compared to the traditional reachability analysis, memory cost of the algorithm is constant to the number of nodes—no state explosion due to the large number of nodes in distributed multi-agent systems.

Finally, we validate the theoretical results on a large distributed sensor network system—Kansei. By applying the theories, ten cases of job interference are detected in an application scenario with two off-line sensing jobs executing in Kansei system (composed of 96 nodes). All interference turns out to be fatal anomalies that make one job overwrite

[1]Note that there are other anomalies not resulting from job interference. We focus on job interference detection and do not attempt to address general anomaly detection in this paper.

978-3-901882-50-0 ©2013 IFIP

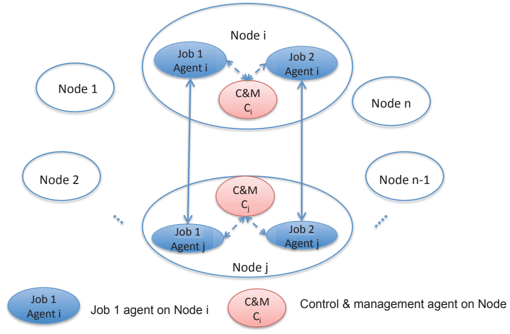


Fig. 1: An Abstract Model of Distributed Multi-Agent Systems

the other. We report our detection to Kansei team and help them diagnose the design flaws.

The rest of the paper is organized as follows. In Section II, the problem is formalized. In Section III, the theorem of the local interference detection of global system operations is derived. The guided random walk algorithm is described in Section III-D. Our case study on Kansei is reported in Section IV.

II. PROBLEM FORMALIZATION

In this section, we first introduce an abstract model of distributed multi-agent systems (Section II-A), then we define concepts about job interference (Section II-B). Some interesting properties of interference detection are shown in Section II-C.

A. The Abstract Model of Distributed Multi-Agent Systems

The models of distributed multi-agent systems are based on communicating extended finite state machines (CEFSMs)—a collection of extended finite state machines (EFSMs) that are interacting with each other [7] . Each EFSM can be expressed as input symbols, output symbols, states, variables, and transitions. In this work, we adopt weak fairness for execution of CEFSMs, that is, if a transition is enabled it can be taken with unbounded number of times.

Consider a multi-agent system with n nodes and m jobs (Fig. 1). Each job runs on a subset of nodes. The jobs on the same node are coordinated by the control & management components in the system. The jobs and the control & management components are modeled by CEFSMs which we shall describe next.

1) Control & Management Components: In practice, control & management components are often job schedulers, resource managers, and data collectors. They can send and receive messages to the job machines on the same nodes. Also they can communicate with the control & management machines on the other nodes. As a general model, the control & management machines on each node does not need to be identical.

2) Jobs: As we described previously, a job executes on a subset of nodes in a multi-agent system. Similar to the control & management components, job machines on two nodes can be different. A job machine can send and receive messages with other job machines regardless whether they are on the same node and whether they are belong to the same job.

3) Variables: CEFSMs variables can be shared among CEFSMs; they model the shared resources and the environment.

4) Channels & Messages: Channels between CEFSMs are assumed to be flawless and synchronized. When CEFSMs execute, they generate inputs and outputs—messages that can be observed in a multi-agent system. We shall assume that the messages transmitted in each channel are mutually exclusive. (In practice, this is achieved by encoding each machine's identification into messages [8].) In this paper, we will use the terms *message* and *IO* alternatively.

5) Job IOs (Messages): For a job, say Job i, there is a set of IOs it is concerned about. Denote the set as IO_i. In practice, if Job i is to control a light in a room, IO_i includes IOs that set the light on/off. Note that IO_i can be sent/received by machines other than Job i machines. Job IOs are specified by system designers before analysis of job interference.

B. Job Interference in Distributed Multi-Agent Systems

A single job is assumed to be well designed. We are curious whether jobs simultaneously running in the same multi-agent system would step upon each others.

Following we first briefly introduce the basic concepts of execution traces and reachability graphs. After that, we show how to project a trace over a single job, or over a single node. Independency and interference are then formally defined. Properties and possible misconceptions are presented finally.

1) Traces and Reachability Graphs: Given a set of CEFSMs, the Cartesian product of states in each CEFSMs forms the set of possible global states. However not all of them are reachable. If one takes transitions from the initial states, a subset of states can be reached. The subset of states as well as the interconnecting transitions generates the *reachability graph*.

In an execution of the CEFSMs, we follow a path from an initial state in the reachability graph and produce IOs as we move step by step. The sequence of IOs is called an *(execution) trace*. For example, a trace may look like "*send_job_files, receive_job_files, submit_results, receive_results, job_terminates*". We denote the set of traces in a reachability graph R as $L(R)$. For simplicity, in the rest of the paper we will use R when we mean $L(R)$.

2) Projection of Jobs: When we study job interferences, we are interested in the behaviors of a single job when it executes with other jobs. The following definition captures the idea.

Definition 1 (Projection of a trace over Job i: $\pi_i(\cdot)$): The function $\pi_i(\cdot) : a \rightarrow b$ is defined over a trace $a = \alpha_1\alpha_2 \ldots \alpha_t$, such that $b = \pi_i(a) = \beta_1\beta_2 \ldots \beta_t$, where

$$\beta_j = \begin{cases} \alpha_j & \alpha_j \in IO_i \\ \lambda \text{ (the empty IO)} & \alpha_j \notin IO_i \end{cases}$$

Example: Job 1 controls lights and Job 2 controls fans. They generate a trace "*light_on, fan_off, light_off*". In view of

the Job 1, the projected trace is "*light_on, light_off*"since the status of fans is not what Job 1 is concerned of.

The definition can be easily extended to projection of a set of traces over Job i and projection over a set of jobs (skipped here).

3) Projection of Nodes: To capture all jobs' behaviors on a single node, we can project execution traces over a node.

Definition 2 (Projection over a node $\pi_{N_j}(\cdot)$): The function $\pi_{N_j}(\cdot) : a \rightarrow b$ is defined over a trace $a = \alpha_1\alpha_2\dots\alpha_t$, such that $b = \pi_i(a) = \beta_1\beta_2\dots\beta_t$, where

$$\beta_j = \begin{cases} \alpha_j & \text{the sender or receiver of } \alpha_j \text{ on Node j} \\ \lambda & \text{else} \end{cases}$$

Example: Projecting the trace "*send_job_to_node1, send_job_to_node2, receive_results_from_node1*" over Node 1 is "*send_job_to_node1, receive_results_from_node1*".

Similarly the definition can be extended to projection over a set of nodes.

The projection operations are commutative.

- Projection on Job i and Job j: $\pi_i \circ \pi_j(\cdot) = \pi_j \circ \pi_i(\cdot)$;
- Projection on Node i and Node j: $\pi_{N_i} \circ \pi_{N_j}(\cdot) = \pi_{N_j} \circ \pi_{N_i}(\cdot)$;
- Projection on Job i and Node j: $\pi_i \circ \pi_{N_j}(\cdot) = \pi_{N_j} \circ \pi_i(\cdot)$.

4) Job Interference and Independency: Start from the simplest case including only two jobs. Intuitively, we compare Job 1's behaviors when it executes alone and the behaviors when Job 1 and Job 2 execute simultaneously. If they are different, Job 1 has *interference* with Job 2; Otherwise, Job 1 is *independent* with Job 2.

Definition 3 (Independency): In a distributed multi-agent system, Job J_1 is *independent* with Job J_2 (denoted as $ind(J_1, J_2)$) if and only if

$$R(\{J_1\}) = \pi_1 \circ R(\{J_1, J_2\})$$

where $R(\{J_1\})$ are the traces when only J_1 executes in the system, and $R(\{J_1, J_2\})$ are the traces when both J_1 and J_2 execute.

If there is such a trace l that makes Job 2 interact with Job 1, we call l an *interference trace*. Our task is to detect the interference traces with soundness and completeness, and without state space explosion.

Extend the previous definition to more-than-two-job scenarios: a system is *interference-free* if and only if for every job J_i in $\mathbf{J} = \{J_1, \dots, J_m\}$, J_i is *independent* with $\mathbf{J}$.

C. Remarks

Before start detecting job interference, we would like to clarify some possible misconceptions.

- Independency is *not* symmetric.
  $ind(J_1, J_2) \nRightarrow ind(J_2, J_1)$.
- Independency is *not* transitive.
  - $ind(J_1, J_2) \wedge ind(J_2, J_3) \nRightarrow ind(J_1, J_3)$
  - $ind(J_1, J_2) \wedge ind(J_1, J_3) \nRightarrow ind(J_1, \{J_2, J_3\})$
  - $ind(J_1, J_3) \wedge ind(J_2, J_3) \nRightarrow ind(\{J_1, J_2\}, J_3)$
- Interference raises an anomaly alarms in distributed multi-agent systems, but it does *not* mean a definite anomaly.
  For example, if Job 2 generates a new IO "*light_off*" when the light is already off, it is interference, but not an anomaly.

Interestingly, we find that interference analysis is scalable to the incremental updates of a system. If a new job J' is added to an interference-free system (with old jobs $\mathbf{J}$), verifying $ind(J', \mathbf{J} \cup \{J'\})$ and $ind(\mathbf{J}, \mathbf{J} \cup \{J'\})$ can guarantee that the new system is also interference-free (shown by Proposition 1).

Proposition 1: For two job sets $\mathbf{J_1} \subseteq \mathbf{J_2}$,
$ind(\mathbf{J_1}, \mathbf{J_2}) \wedge ind(\mathbf{J_2}, \mathbf{J_3}) \implies ind(\mathbf{J_1}, \mathbf{J_2} \cup \mathbf{J_3})$.

III. DETECTING JOB INTERFERENCES

In this section, we derive the theorem of the local interference detection. A guided random walk algorithm is then proposed for obtaining local observations.

A. State Explosion

Following the definition of dependence, one can detect job interference by constructing two reachability graphs—one with a single job executing and the other with multiple jobs executing simultaneously—and comparing the behaviors of a targeted job based on the global observations.

However, state explosion makes the straightforward approach infeasible in distributed multi-agent systems, which usually comprise a large number of nodes. For example, a reachability graph of a 100-node system with 10 states on each node can reach 10^{100} states in worst case. In our case study Kansei, a reachability graph including only 14 nodes already runs out of the 512M Java Virtual Machine heap memory.

B. Can Local Interference Detection Replace the Global One?

Global observations generate the troubles. An intriguing question is thus asked.

Question 1: Can we detect job interference locally, that is, if we do not observe interference on any node, can we declare that the whole system is interference free? Formally,

$$\forall N_j,\ \pi_{N_j} \circ R(\{J_1\}) = \pi_{N_j} \circ \pi_1 \circ R(\{J_1, J_2\}) \iff R(\{J_1\}) = \pi_1 \circ R(\{J_1, J_2\})?$$

The "$\Leftarrow$" direction is the soundness: whether a local detected interference trace is a global one; The "$\Rightarrow$" direction is the completeness: whether all job interference can be observed locally. Our study focuses on two-job scenario. The conclusions can be extended to multi-job cases.

1) Soundness: It is shown by Lemma 1.

Lemma 1: A local detected interference trace is a global one.

For a distributed multi-agent system with Job J_1 and Job J_2, if the projection of $R(\{J_i\})(i = 1, 2)$ over a node, say Node j, is different from the projection of $R(\{J_1, J_2\})$ over Node j and Job J_i, then J_i has interference in the system. (For space limitation, we skip the proof.)

2) Completeness: The completeness means that any global interference can be detected locally. It includes two parts:

- *Part 1*: If a trace exists when two jobs execute simultaneously, but it does not exist when a job executes alone, the trace can be detected locally (shown by Lemma 2);
- *Part 2*: If a trace exists when a job executes alone, but does not exist when two jobs execute simultaneously, it can be detected locally.

Lemma 2: If a trace exists when two jobs execute simultaneously, but does not exist when a job executes alone, the trace can be detected locally.

That is to say, if there is a trace l in the projection of $R(\{J_1, J_2\})$ on Job i, but *not* in $R(\{J_i\})$, then there exists a Node j, the projection of $R(\{J_i\})$ on Node j is different from the projection of $R(\{J_1, J_2\})$ on Node j and Job J_i.

However, unlike Part 1, Part 2 is not always true. Following is a counterexample: a system has two jobs (J_1 and J_2) and two nodes (N_1 and N_2). A variable x (initial value 0) is shared by the two jobs on N_2. J_1 has the logic: on N_1 it always outputs a, b, c; on N_2 it outputs $1, 2, 3$ if $x = 0$. When J_1 executes alone, the global observations include all interleaves of abc and 123. J_2 has the logic to change x to 1 at the beginning and reset $x = 0$ if it sees all three outputs a, b, c. When J_2 is present and scheduled first, J_1 can output $1, 2, 3$ only if a, b, c have been outputted—an interference, but neither of two nodes can detect it locally.

The counterexample shows that completeness does not always hold, as disappearance of traces may not be detected locally. We thus ask Question 2.

Question 2: In what kinds of multi-agent systems, one can locally detect the traces that exist when a job executes alone, but not exist when two jobs execute simultaneously?

Lemma 3: For a distributed multi-agent system that does not have shared variables or control & management components, local detection can identify the traces existing when a job executes alone, but not when two jobs execute simultaneously (see the proof in appendix).

That is to say, if there is an l in $R(\{J_i\})$ and but *not* in the projection of $R(\{J_1, J_2\})$ on Job i, there exists Node j, such that the projection of $R(\{J_i\})$ on Node j is different from the projection of $R(\{J_1, J_2\})$ on Node j and Job i.

The systems satisfying the above conditions are limited, as most multi-agent systems have control & management components. Before we relax the restrictions, the concept of *well-extension* is defined.

Definition 4: For a trace l in the system when Job J_i executes alone, denote its projection on machine M as l_M. Among traces generated when two jobs execute, if any one whose projection on M and J_i is a prefix of l_M can be extended to a trace whose projection is l_M, then machine M is *well-extended* for Job J_i.

Example: when Job 1 executes alone, it produces a trace whose projection on machine M is "*light_on, light_off*". When Job 1 and Job 2 executes, project all the traces on M and Job 1. Consider the traces whose projection is "*light_on*". If any of them has an extended trace whose projection is '*light_on, light_off*', then M is well-extended for Job 1.

Intuitively, the well-extended property states that, comparing the projections of two reachability graphs—one generated when a single job runs and the other generated when multiple jobs run—on a specific job and a specific control & management machine, the former is a subgraph of the latter.

We now extend the answer to Question 2.

Lemma 4: For a distributed multi-agent system without shared variables, if all the control & management machines are well-extended for every job, local detection can identify the traces existing when a job executes alone, but not existing when two jobs execute simultaneously (proved in appendix).

One may ask why the property is needed and whether it is widely satisfied. Here are some insights. Intuitively, the well-extended property says that, if a projected trace does not happen in the multi-job scenario, it does not happen in the single-job scenario. The well-extended property can rule out a class of interference: a projected trace happens in a single-job scenario, but not in the multi-job scenario. An interference-free system always satisfies the well-extended property.

To further remove the restriction on shared variables, we next introduce the concept of *fine-grained specification*.

Definition 5: A shared variable has a *fine-grained specification* if any transition that reads/writes the variable generates a message showing the value it reads/writes. If the variables shared among different jobs or among jobs and control & management machines have fine-grained specifications, the system model is *fine-grained*.

Example: The shared variable x in the previous counterexample does not have a find-grained specification—Job 2 can modify x's value without generating a message.

With the concept of find-grained model, Lemma 5 further extends the answer to Question 2.

Lemma 5: If all the control & management machines are well-extended for every job and the system model is fine-grained, local detection can identify the traces existing when a job executes alone, but not existing when two jobs execute simultaneously.

Again, one may question whether the fine-grained property is common in distributed multi-agent systems. It is a misconception, as the property is a requirement of modeling rather than that of a system. Once a model captures read/write operations on variables shared either by two different jobs or by a job and its control & management component, we call the model fine-grained.

From Lemma 1 to Lemma 5, we can have Theorem 1.

Theorem 1: For a distributed multi-agent system, if the control & management machines are well-extended for every job and the system model is fine-grained, then global job interferences can be detected locally; it is sound and complete.

C. Remarks

First, local observations are not the observation of *one* system run, but the *set* of all local traces in all the system runs. It is easy to obtain *a* local trace, but covering *all* of them is nontrivial.

Second, one may feel the localization of detection is against intuitions if he regards interference detection the same with anomaly detection. These two tasks are different, and job interference is an important source of anomalies in distributed multi-agent systems where independent jobs execute.

D. Random Walk Guided by Local Observations

Following we propose a *guided random walk algorithm* (Algorithm 1) to cover the local observations of Job i at Node j without constructing the global reachability graph.

Algorithm 1 Random Walk Guided by Local Observations

Input:
The examined system, $\mathbf{T}(\mathbf{M}, \mathbf{V}, \mathbf{IO}, S_0, V_0, m, n)$;
The examined job, Job i; The observed node, Node j;
The number of rounds, nr; The depth of a round, d;
The guiding vector, $\mathbf{G}$;
Output:
The set of covered local traces of Job i at Node j, H;

```
1: H ← ∅, init(S0, V0) {Initialization}
2: for round = 1 to nr do
3:   ct ← null {Initialize the current trace ct}
4:   for depth = 1 to d do
5:     for all m ∈ M do
6:       W(m)← 0{Initialize the weight vector}
7:       if isExecutable(m) = True then
8:         add G.WeightOfMachine(m) to W(m)
9:       if isJobiMachine(m) = True then
10:        add G.WeightOfJobM(m) to W(m)
11:      if isOnNodej(m) = True then
12:        add G.WeightOfNodej(m) to W(m)
13:      if isNewTransition(m) = True then
14:        add G.WeightOfNewIO(m) to W(m)
15:      if isExternalJobiIO(m) = True then
16:        add G.WeightOfExtJobiIO(m) to W(m)
17:    W← normalize(W); nt ← random(M,W)
18:    if nt = null then
19:      break {No machine to execute}
20:    else
21:      executeOneStep(nt)
22:      update(ct) {Update the current trace}
23:  update(H,ct) {Update covered traces}
24: return H
```

The algorithm contains three layers of loops. The outer loop and the second inner loop conduct the random walk with nr rounds (at most d steps in each round). In each step, the inner most loop computes the transition probability based on the guiding vector $\mathbf{G}$ (Line 5-Line 16). A machine is thus randomly picked and executed.

The entries in the guiding weight vector are as follows. *Weight of Machines* describes the relative speed of each machine. If the server machines have lower weight than the client machines, interference due to heavy load on server will be discovered early. *Weight of Job i Machines* describes the possibility to pick Job i machines. A low value favors the local observations with the possible interferences of Job i. *Weight of Node j* describes the possibility of picking machines on Node j. A large number is preferred to quickly cover more traces on observed node. *Weight of New Job i Transitions* is preferred to be large, as it describes the possibility of choosing a machine that generates a new Job i symbol in the next step. *Weight of External Job i IOs* describes the possibility of picking a non-job i machine that can generate a Job i IO in next step. A large value is preferred, because these IOs usually generate interference.

If a Job i machine M_k is picked with probability p_k and takes s_k steps to finish the execution, then we set the round depth as $d = 10 \times max_k(\frac{s_k}{p_k})$. According to the Chebyshev's inequality, d steps can guarantee that 90% of the rounds finish execution of Job i.

The memory cost of the algorithm is $O(|H_{i,j}| + d)$, where $|H_{i,j}|$ is the size of all local observations of Job i at Node j. The memory consumption is independent of the number of nodes and hence our algorithm is scalable.

The time cost of the algorithm is $O(nr \times d \times |\mathbf{M}| \times |H_{i,j}|)$, where nr is the number of rounds, $|\mathbf{M}|$ is the number of machines. $|H_{i,j}|$ is a multiplication factor because in each step we need to check for each machine whether the next transition generates a new local observation (Line 13). The random walk algorithm can cover *all* local observations of Job i on Node j with high probability when the number of rounds is large enough (*e.g.* the number of global traces). The time cost can be exponential to the number of nodes–the same with the straightforward approach.

Theorem 1 transfers the high probability of the coverage to the high confidence of global interference knowledge. Without Theorem 1, one cannot claim his confidence no matter how many local traces are observed, since the interference may be just invisible locally.

IV. Case Study: Kansei

A. The Architecture of Kansei

The Kansei testbed [9], [1] is designed to facilitate research on sensor network applications in a large scale. It provides a testbed for conducting wireless sensor network experiments on various wireless platforms as well as diverse sensor node platforms, including XSM, Telosb, iMote2 and Stargates. Kansei 1.0 consists of 96 Kansei Nodes. As shown in Fig. 2, each Kansei node comprises of a Starget and three attached sensors. The job control and management components reside on the central Kansei Director and each Stargate.

Kansei specifications are modeled by a set of CEFSMs (details skipped due to the space limit).

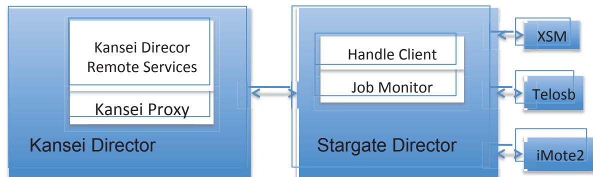


Fig. 2: Kansei Architecture

B. Properties of Kansei

Kansei model satisfies the two properties: (1) the model is fine-grained; and (2) the control & management components are well-extended.

1) Fine-grained: The shared variables represent the sensor hardware shared by jobs. We modele its read/write operations with distinguished messages. Therefore the model is fine-grained.

2) Well-extended: We study control & management machines in Kansei model one by one. (1) The Time machine sends Job i start-time and end-time messages in a fixed order, independent of other jobs. Time machine is well-extended. (2) The KanseiDirectorRemote machine receives time messages and sends START messages to HandleClient machines. The START messages can be sent if and only if the start-time message is received. If no local interference is found, no other Job i message will be generated. The machine is thus well-extended. (3) KanseiProxy only receives messages FileSt and FileFin. FileSt messages are always accepted. FileFin can be accepted only if its corresponding FileSt is at the head of a queue (FileSt is removed after the acceptance). Therefore KanseiProxy receiving Job i's FileSt from Node j eventually leads to receiving the corresponding FileFin—independent of other jobs. KanseiProxy is well-extended. (4) A JobMonitor machine receives Job i's end-time messages from the Time machine and sends out its FileSt and FileFin. It is well-extended. (5) A HandleClient machine receives a START message from the KanseiDirector and sends out the start signal to a sensor mote—well-extended too. In summary, the control & management components of Kansei satisfy the well-extended property.

With the two properties, global interference detection on Kansei can be replaced by the local one.

C. Interference Detection

Before jobs are submitted to Kansei, one detects job interference to avoid anomalies resulted from the interference. We study a case where two off-line sensing jobs are submitted.

1) Settings: Considering the scenario where Job 1 is scheduled on Node 0–71 and Job 2 is on Node 0–45 and Node 72–95. They only collect data on XMS sensors and send back to the server. Job 1 is supposed to start on Time 1 (logical time) and finish on Time 2 while Job 2 is to start on Time 3 and finish on Time 4.

2) Interference Traces Detected: The guided random walk algorithm was applied on the Kansei model. we chose the depth of walk as 200, the weight of Time machine as 4, the weight of KanseiDirectorRemote machine as 80, the weight of Job 1 machines as -0.9, the weight of new Job 1 transitions as 50, and the weight of external Job 1 IO as 100. Due to the symmetry, the overlapped nodes of two jobs have the same local observations. We thus focus on Node 0. Similarly, Job 1 and Job 2 are symmetric, and we thus focus on detecting interference on Job 1.

TABLE I: Local Interference Traces Detected

1	t=1	t=2	j1St	j2St	End		
2	t=1	j1St	t=2	j2St	End		
3	t=1	t=2	j1St	j2St	j1FileSt	End	
4	t=1	j1St	t=2	j2St	j1FileSt	End	
5	t=1	t=2	j1St	j2St	j1FileSt	j1FileFin	End
6	t=1	j1St	t=2	j2St	j1FileSt	j1FileFin	End
7	t=1	t=2	j1St	j1FileSt	j2St	End	
8	t=1	j1St	t=2	j1FileSt	j2St	End	
9	t=1	t=2	j1St	j1FileSt	j2St	j1FileFin	End
10	t=1	j1St	t=2	j1FileSt	j2St	j1FileFin	End

After running 100,000 rounds, 19 local traces were observed on Node 0. According to the Job IOs, they are at most 19 local traces—we covered all local traces.

Comparing the observed local traces with the Job 1 traces when it is executed alone, we identified ten interference traces (shown in Table I). These traces only exist when Job 1 and Job 2 run in the system, but not when Job 1 runs alone. Take trace 5 as an example. The interference trace includes six job IOs: *time 1, time 2, start of Job 1, start of Job 2, start of Job 1 file submission,* and *finish of Job 1 file submission*. We can see that Job 2 starts before the Job 1 submits files. Therefore, Job 2 will be programmed to the sensor while Job 1 is still running—the interference actually leads to a severe anomaly. The first six interference traces all generate the same anomaly. In the remaining four traces, Job 1 finishes execution, but it cannot finish data transmission due to the interruption of Job 2. The interference also produces an anomaly in Kansei system. We will further discuss the roots of interference in Section IV-D.

3) Memory Consumption: To measure memory performance of the proposed algorithm, the algorithm was evaluated on several cases: starting from a 4 node scenario where each job uses 3 nodes with 2 overlapped nodes, and increasing the number of nodes to 96; The experiments were repeated with the number of rounds from 10 to 10,000. The heap memory consumption in Java Virtual Machine was measured.

Fig. 3a shows that as the number of nodes increases, the average memory consumption remains stable regardless the number of rounds. The result is consistent with the theoretical analysis in Section III-D. In contrast, when we constructed the whole reachability graph, the 512M heap memory of Java virtual machine quickly ran out with only 14 nodes.

Fig. 3b shows that the memory consumption is stable as the number of covered local traces increases, although the previous theoretical analysis indicates a linear increase. It is because the memory consumption of new collected trace is shadowed by the large memory cost on CEFSM construction and temporary system state maintenance, which are constant to the number of covered traces. The results do not change when the number of nodes varies.

4) Coverage Time: The coverage time was also analyzed. We used the number of rounds as a metric and observed how the local traces were gradually covered in the random walk with 100,000 rounds. The experiments were repeated with the 53 node scenario and 96 node scenario. The results are shown in Fig. 3c. In both scenarios, the first several traces are quickly covered due to the guiding vector. As the number of covered traces increases, the number of rounds to cover the next new local trace grows exponentially. About 100,000 rounds, all 19 local traces are covered. The exponential growth result is consistent with our theoretical analysis.

D. Sources of interferences

The detected interference and the anomalies were reported to the Kansei testbed team. The developers confirmed their existence. With interference traces, we further helped them to diagnose three sources of interference.

First, in Kansei Director, the scheduler fetches jobs in a job table and dispatches them if the current time is later than the jobs' expected start time. Since Director checks the job table every 30 seconds, the successive jobs that have close expected start time may be scheduled out of order.

Second, Kansei Director reserves resources for a job only if it is still running. Because Kansei Director determines a job's status according to their end-time but not its real status, a new job might be programmed to the same node while an old one is still submitting the experiment data.

Third, when a job finishes on a Stargate node, it attempts to upload the log files to Kansei Proxy by a Socket connection. The Kansei Proxy works in a single thread way. It receives connection requests and puts them into a queue. Only when the request is at the head of the queue, data transfer starts. Kansei Director may schedule a new job on the same node while a job machine on a Stargate is still waiting in the queue.

The detected interference as well as their sources indicates two modifications of Kansei system design: (1) The system should enable Kansei Director to know the real status of jobs, so that a new job will not be scheduled until the old one has completed. (2) The single-threaded socket in Kansei Proxy could be replaced with a multi-thread one to avoid the potential congestion in queue.

V. Related Work

Job interference problems were originally studied in *centralized* telephony systems (referred as feature interaction problems) where multiple independent features, such as call forwarding and call waiting, are required to coexist without interference. For example in [10], [11], feature interaction was formally defined; theorems and algorithms were proposed to avoid state explosion due to the large number of features. Our work is inspired by the study, but focuses on avoiding explosion generated by the large number of nodes, typical in distributed systems. In [12], similar interaction problems were studied in the context of distributed SIP call systems, and it was extended to embedded control systems [13] and networked home appliances [2]. However, they did not address the state explosion in large distributed systems either. The scalability problem due to the large number of agents were studied in the context of detecting disagreements among team-members[14], but it did not aim at detecting job interference. Although general anomaly detection has been well studied in distributed systems (*e.g.* [15], [16], [17], [18]), we are not aware of study focusing on job interference detection in large distributed multi-agent networks.

Our work is also inspired by flow security analysis that checks whether a program may leak information about its high (*i.e.* secret) inputs into its low (*i.e.* public) outputs. The conventional method is to enforce *noninterference* such that low outputs are independent of high inputs [19]. Recent work includes noninterference for deterministic interactive programs [20], quantitative analysis of information leak [21], and information flow security in distributed systems [22].

VI. Conclusion

In this paper we studied the job interference detection of distributed and multi-agent systems using a formal approach. To cope with the state explosion problem we obtained sufficient conditions, which allowed us to reduce the interference detection to that on system components, instead of an analysis of global system behaviors. We showed that our method is sound and complete. At each network node, a memory efficient guided random walk algorithm was applied for detecting interference. We applied our method to Kansei wireless sensor network system and uncovered ten new interference traces.

To apply our theory, we require pre-knowledge of system topology. How to address the interference detection problem without a fixed topology is our future work. Also, our theory requires a system to satisfy two conditions: fine-grained and well-extended. They are sufficient conditions. What is the necessary and sufficient condition remains an open problem.

References

[1] A. Arora, E. Ertin *et al.*, "Kansei: A High-Fidelity Sensing Testbed," *IEEE Internet Computing*, pp. 35–47, 2006.

[2] M. Kolberg, E. Magill *et al.*, "Compatibility Issues between Services Supporting Networked Appliances," *Communications Megazine, IEEE*, vol. 41, no. 11, pp. 136–147, 2003.

[3] G. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[4] E. Clarke, "Model Checking," in *Foundations of Software Technology and Theoretical Computer Science*. Springer, 1997, pp. 54–56.

[5] M. Calder and A. Miller, "Feature Interaction Detection by Pairwise Analysis of LTL Properties - A Case Study," *Formal Methods in System Design*, vol. 28, no. 3, pp. 213–261, 2006.

[6] A. Felty and K. Namjoshi, "Feature Specification and Automated Conflict Detection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 1, pp. 3–27, 2003.

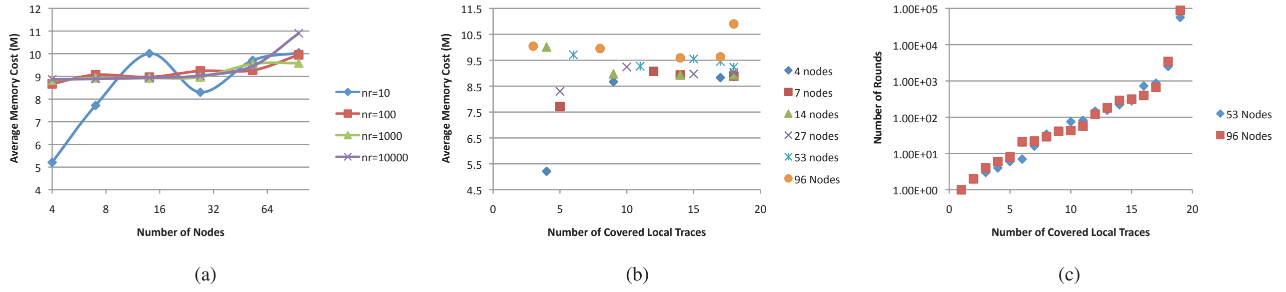
[7] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines - A Survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[8] S. Lam and A. Shankar, "Protocol Verification via Projections," *IEEE Transactions on Software Engineering*, no. 4, pp. 325–342, 1984.

[9] "KanseiGenie," http://kansei.cse.ohio-state.edu/KanseiGenieFed/index.php.

[10] T. LaPorta, D. Lee *et al.*, "Protocol Feature Interactions," in *Proceedings of FORTE-PSTV*, 1998, p. 59.

[11] A. Arcuri and L. Briand, "Formal analysis of the probability of interaction fault detection using random testing," 2011.



(a) (b) (c)

Fig. 3: (a) Average memory cost with varying number of nodes. (b) Average memory cost with varying number of local traces covered. (c) Number of rounds with varying number of local traces covered.

[12] M. Kolberg and E. Magill, “Managing Feature Interactions between Distributed SIP Call Control Services,” *Computer Networks*, vol. 51, no. 2, pp. 536–557, 2007.

[13] A. Metzger, “Feature Interactions in Embedded Control Systems,” *Computer Networks*, vol. 45, no. 5, pp. 625–644, 2004.

[14] G. Kaminka, “Detecting Disagreements in Large-Scale Multi-Agent Teams,” *Autonomous Agents and Multi-Agent Systems*, vol. 18, no. 3, pp. 501–525, 2009.

[15] P. Tichỳ and R. Staron, “Multi-Agent Technology for Fault Tolerant and Flexible Control,” *Innovations in Multi-Agent Systems and Applications-1*, pp. 223–246, 2010.

[16] S. Haegg, “A Sentinel Approach to Fault Handling in Multi-Agent Systems,” *Multi-Agent Systems Methodologies and Applications*, pp. 181–195, 1997.

[17] L. Liu, K. Logan *et al.*, “Fault Detection, Diagnostics, and Prognostics: Software Agent Solutions,” *IEEE Transactions on Vehicular Technology*, vol. 56, no. 4, pp. 1613–1622, 2007.

[18] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM Computing Surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[19] A. Sabelfeld and A. Myers, “Language-Based Information-Flow Security,” *Selected Areas in Communications, IEEE Journal on*, vol. 21, no. 1, pp. 5–19, 2003.

[20] D. Clark and S. Hunt, “Non-Interference for Deterministic Interactive Programs,” *Formal Aspects in Security and Trust*, pp. 50–66, 2009.

[21] G. Smith, “On the Foundations of Quantitative Information Flow,” *Foundations of Software Science and Computational Structures*, pp. 288–302, 2009.

[22] N. Zeldovich, S. Boyd-Wickizer *et al.*, “Securing Distributed Systems with Information Flow Control,” in *5th USENIX Symposium on Networked Systems Design and Implementation*, 2008, pp. 293–308.

APPENDIX

Proof of Lemma 3: WLOG, let $i = 1$. Consider a $l \in R(\{J_1\})$ and $l \notin \pi_1 \circ R(\{J_1, J_2\})$. There are two cases.

Case 1: An anomaly trace l has a new projection on some node that does not appear in $R(\{J_1, J_2\})$, that is, $\exists \pi_{N_j}(l) \notin \pi_{N_j} \circ \pi_1 \circ R(J_1, J_2)$. The anomaly can be detected on N_j.

Case 2: An anomaly trace l does not have any new projection on any node, but the interleaving of projections is new. That is to say, $\forall N_j$, $\pi_{N_j}(l) \in \pi_{N_j} \circ \pi_1 \circ R(J_1, J_2)$, but $l \notin \pi_1 \circ R(\{J_1, J_2\})$. If there is such a l, the anomaly cannot be detected locally. We show that this kind of l does not exist.

Because there is no control & management part or shared variables, J_1 machines can only be affected by J_2 machines via inter-job messages. Since the message set sent in channels are exclusive. Inter-job messages between J_2 and J_1 machines do not appear in $R(\{J_1\})$ (where only J_1 machines exist), that is to say, inter-job messages are not in l. Hence in trace l, Job 1 is not affected by J_2, *i.e.* l is not an anomaly trace. It is contradicting to the condition. □

Proof of Lemma 4: WLOG let $i = 1$. Similar to Lemma 3, the proof is two-folded. On the one hand, if $\exists \pi_{N_j}(l) \notin \pi_{N_j} \circ \pi_1 \circ R(J_1, J_2)$, the anomaly can be detected on N_j. On the other hand, if an anomaly trace l does not have any new projection on any node, but the interleaving of projections is new, l cannot be locally detected. Since no new projection is observed, l does not include any messages sent and received by J_2 machines. Assume $l = \alpha_1\alpha_2 \dots \alpha_t$, now we show that l can be constructed in $R(\{J_1, J_2\})$ by induction. In other words, l is not an anomaly trace.

Base: Consider α_1. Because local observations are the same, α_1 can be generated.

Hypothesis: Assume $\alpha_1\alpha_2 \dots \alpha_k$ can be constructed.

Induction: Consider $\alpha_1\alpha_2 \dots \alpha_k\alpha_{k+1}$

Case 1: α_{k+1} is a message sent or received by J_1 machine at N_j. $\alpha_1\alpha_2 \dots \alpha_k$ has been generated, so does its projection on Node j: $l_j = \pi_1 \circ \pi_{N_j}(\alpha_1\alpha_2 \dots \alpha_k)$. Since any message sent or received by J_1 machines will be observed, l_j includes a trace of J_1 machines on Node j. According to the observation on $R(\{J_1\})$, the trace l_j of J_1 machines on Node j can be extended by α_{k+1}. Because there are no shared variables or messages sent by J_2 in $R(\{J_1, J_2\})$, J_1 is able to send or receive α_{k+1} based on what it received in $\alpha_1 \dots \alpha_k$.

Case 2: α_{k+1} is a message sent or received by control & management machines at N_j. In $\alpha_1\alpha_2 \dots \alpha_k$, consider J_1 IO sequence sent or received by control & management machines (denoting it as l_c). In $R(\{J_1\})$, it is observed that l_c can be extended by α_{k+1}. Since the control & management machines on N_j are well-extended and the local observations are the same, in $R(\{J_1, J_2\})$, l_c can also be extended by α_{k+1}.

Above all $l \in \pi_1 \circ R(\{J_1, J_2\})$, a contradiction. □