On the Computability of Agent-Based Workflows

By: Wai Yin Mok, Prashant Palvia, and David Paper.

Mok, W.Y., Palvia, P., and Paper, D. (2006). "On the Computability of Agent-Based Workflows." *Decision Support Systems*. 42, (3), 1239-1253.

Made available courtesy of Elsevier: http://dx.doi.org/10.1016/j.dss.2005.10.010

Abstract:

Workflow research is commonly concerned with optimization, modeling, and dependency. In this research, we however address a more fundamental issue. By modeling humans and machines as agents and making use of a theoretical computer and statecharts, we prove that many workflow problems do not have computer-based solutions. We also demonstrate a sufficient condition under which computers are able to solve these problems. We end by discussing the relationships between our research and Petri Nets, the multi-agent framework in the literature, linear programming and workflow verification.

Keywords: Undecidability | Agents | Statecharts | Turing machines | Abacus programs | Workflows | Capabilities

Article:

1. Introduction

This paper investigates if computer-based decision support systems are able to solve the following research question of interest: Is a particular group of agents (humans and machines) able to achieve a business goal that is defined in terms of producing a certain number of units of a quantifiable output resource? This research question brings up a very fundamental issue. By modeling humans and machines as agents [13], we shall prove that computers cannot solve our research question, or our research question is undecidable [16], unless we restrain it in some ways. The argument of this paper is based on several assumptions. First, we define an agent framework such that no two distinct agents can execute simultaneously (see Section 3.1) and an agent can at most call on two other agents to perform work (see Section 3.2). Nevertheless, this framework has no loss of generality since it is general enough to model abacus programs, which have the same computational power as Turing machines (see Theorem 2 in Section 3.2). Second, only quantifiable resources are considered (see Assumption 1 in Section 3.2). Third, all human jobs must be simple and mundane such that they all can be completely characterized as

algorithms (see Assumption 2 in Section 3.2). Fourth, time taken by a group of agents is not considered at all. Such a group may take as much time as it wants to complete its workflow (see Section 3.1). These assumptions obviously restrict this research to a narrow class of agent-based workflows. However, we shall prove in Section 3 that computers cannot even solve our research question for this narrow class of agent-based workflows. Then any broader agent framework which subsumes our framework as a special case will certainly have the same computational problems presented in Section 3 as well (see Section 4.2).

The implications of this paper on workflow designs are as follows. Since business process reengineering has claimed many successes in recent years [5] and [6], a company might redesign its workflows for more efficiency. Since our research question is fundamental for the correctness of a workflow, any workflow designer must answer it. Nevertheless, by Theorem 3 and Theorem 4, there is no computer-based decision support system that is able to solve our research question. On the other hand, by Theorem 5 and its corollaries, if every resource required by a workflow is bounded, then computers become able to solve our research question. However, Theorem 6 shows that determining the bound for a resource required by a workflow is undecidable itself. Therefore, sooner or later human decisions must be made in a workflow design.

A word of caution is appropriate concerning this conclusion. We do not mean that computers are of no use for designing workflows. However, we do mean that designing workflows is not an exact science and many guesses have to be made during the design process. As our techniques become more accurate, most likely assisted by computers, we may be able to devise better and better workflow designs. However, our research question can never be solved *completely* no matter how sophisticated our techniques become and human decisions, which in many cases are simply guesses, must be made.

In order to prove that our research question does not have a computer-based solution, we must prove that it does not have an algorithmic solution. For this purpose, we need a formal definition of algorithms. The belief that Turing machines are sufficient to define algorithms is known as the Church–Turing Thesis. Although the Church–Turing Thesis cannot be proved because there are unlimited computational models, most mathematicians believe that the Church–Turing Thesis is true [3] and [16]. In this research, we make use of an equivalent computational model, namely abacus programs, to prove that some agent-based workflow problems are inherently unsolvable by computers.

Turing machines define algorithms formally. However, they are too low-level in nature to specify various features of workflows. As part of the Unified Modeling Language (UML), Harel's statecharts are used to model the reactions of a system when it faces external stimuli [2]. Moreover, statecharts have been used to model various workflow concepts specified by the Workflow Management Coalition [11]. Consequently, we chose statecharts as a vehicle to prove the assertions we make in this research. As an example, Fig shows a statechart model of a workflow in a simple garment factory. The states, which are represented as round-cornered rectangles in Fig. 1, show the basic activities needed in the workflow. The transitions, represented as arrows, are optionally guarded by conditions. Whether transitions will take place depend upon the truth and falsity of such conditions.

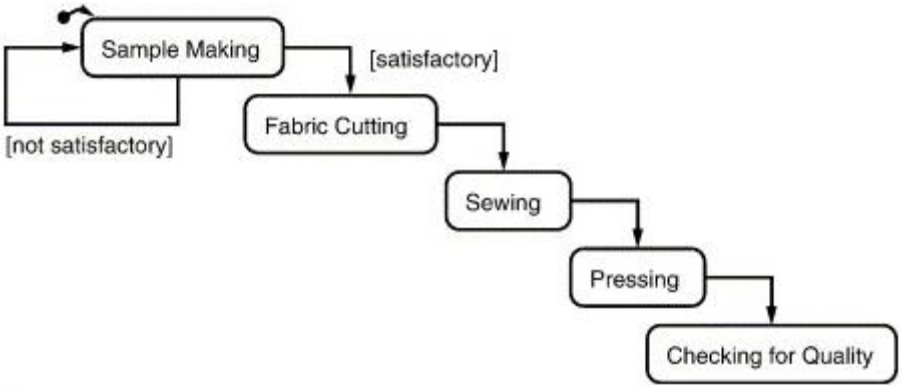

Fig. 1.
A workflow in a simple garment factory.

This paper is organized as follows. In Section 2, we establish the halting problem of statecharts and prove as corollaries some of its consequences which will be used later in the paper. Section 3 proves Theorem 3 and Theorem 4–the main results of this paper–which collectively provide a negative answer to our research question. In addition, Section also proves Theorem 5 and its corollaries, which show that if every resource required by a workflow is bounded, then our research question becomes decidable. Section discusses the relationships between this work and Petri Nets [12], the multi-agent framework in [15], linear programming [9] and workflow verification. The main ideas and implications are summarized in Section 5.

2. The halting problem of statecharts

An *abacus* is a theoretical computer in the sense that it has an unlimited number of registers and each register can store a number of any size [3]. No real computers have these two properties since a real computer has a fixed number of memory cells and each memory cell can only store a number up to a certain number of digits. Since we are studying the theoretical aspects of statecharts, it is not inappropriate to ignore these physical limitations. Registers in an abacus are like elements in an array of a programming language. Thus, we use a similar notation reg(i) to denote the ith register in an abacus, where $i \geq 1$. Since other kinds of objects such as strings and negative integers that can be manipulated by computers can be encoded and decoded as nonnegative integers [16], in this paper we only consider nonnegative integers. Consequently, numbers stored in registers are integers greater than or equal to zero.

An abacus can add one to a register and subtract one from a register if the current number in the register is greater than zero. Since no other operations are possible with an abacus, (in this sense) it is a very primitive computer. Graphically, these additions and subtractions are represented as nodes, as shown in Fig. 2. The intended operation of a node is written as its label. Because negative numbers are not allowed, there are only two possibilities with subtractions: either the number in a register is already zero or is greater than zero. In the first case, the intended subtraction is ignored and the outgoing arrow marked with *e* is followed to locate the next operation. [1] In the second case, one is subtracted from the register and the outgoing arrow *not* marked with *e* is followed to find the next operation. Note that any outgoing arrows in Fig. 2 can be absent. For example, if the outgoing arrow of a node of addition is missing, no more operations will be executed after the addition is done. Similarly, either one or both of the

two outgoing arrows of a node of subtraction can be missing. An *abacus program* is a directed graph of these nodes and arrows. One of the nodes in a program is specified as the start node, which means its operation is the first one to be executed.

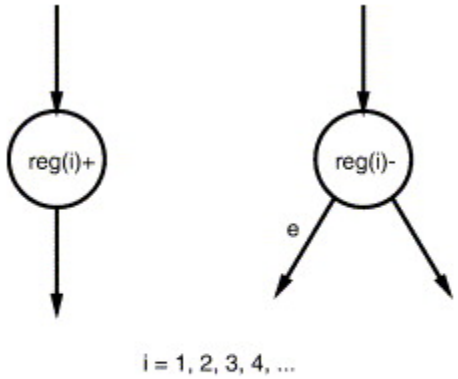

Fig. 2.
Two elementary operations of an abacus.

Following the additions and subtractions of an abacus program, input values are transformed into output values. In this sense, abacus programs compute mathematical functions. Given a mathematical function *f*, if there exists an abacus program that computes *f*, *f* is called an *abacus-computable* function. Specifically, an abacus program *P* computes an *n*-place function $f(x_1, \ldots, xn)$ as follows. Given *n* nonnegative integers $x_1, \ldots, xn$ ($n \geq 0$) as input values,[2] reg(1) is set to $x_1, \cdots$, reg(*n*) is set to *xn*, and all other registers are set to zero. *P* then starts executing. When *P* halts, reg(1) is the result of the computation. The numbers in all other registers are simply ignored. That is, $f(x_1, \ldots, xn)$ = reg(1) if *P* halts. If *P* never halts, $f(x_1, \ldots, xn)$ is undefined. For example, Fig. 3 shows an abacus program that computes the minimum of three given nonnegative integers, where the leftmost node is the designated start node.

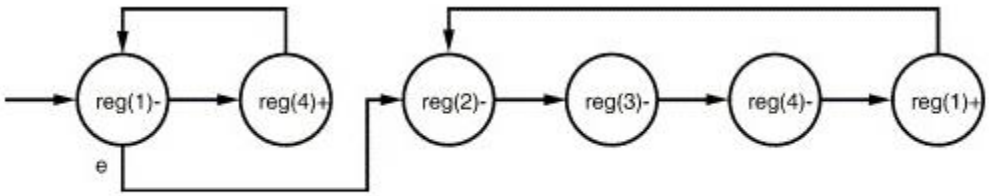

Fig. 3.
An abacus program that finds the minimum of three numbers.

The Church–Turing Thesis implies that the halting problem of Turing machines is undecidable (see page 49 in [3]). That is, there does not exist any algorithmic solution to decide if a Turing machine will halt or not after it starts computing a function. This result has important implications on abacus programs, which are listed here.

1. Because Turing computable functions are abacus computable, abacus programs are able to model Turing machines (see Chapters 6, 7, and 8 in [3]). Since the halting problem of Turing

machines is undecidable, therefore the halting problem of abacus programs is also undecidable (see page 66 in [3]).

2. The halting problem of any formalism that is able to model abacus programs is then undecidable.

Abacus programs have the same computational power as Turing machines. This means that abacus programs, like Turing machines, can also capture the very essence of algorithms. However, a human agent or a machine can do much more than adding one to or subtracting one from a register. For this reason, we need to choose another modeling tool, or a formalism, that allows us to more naturally model what an agent is able to do. Although we can use any formalism that is able to model abacus programs to establish the results in this research, our choice is statecharts [7]. We now present a fundamental result of statecharts.

Theorem 1.

Whether a statechart will halt or not after it starts executing is undecidable.

Proof.

See Appendix A.

Theorem 1 leads to several important consequences, which are proved as Corollary 1 and Corollary 2 below. Corollary 1 and Corollary 2 consider a particular set P of abacus programs. An abacus program *P* is a member of P if *P* has some nodes with missing outgoing arrows. If an abacus program *P* is in P, it does not necessarily mean *P* will halt. However, if *P* will halt, *P* must be a member of P. In contrast, if an abacus program *P* is not in P, we can immediately conclude that *P* will not halt for all possible input values.

Corollary 1.

Whether a particular state will be entered in a statechart is undecidable.

Proof.

See Appendix A.

Corollary 2.

Whether a particular nontrivial condition will become true in a statechart is undecidable. (Note that a nontrivial condition is neither always true nor always false.)

Proof.

See Appendix A.

3. Agent-based workflows

We shall prove in this section that in general the research question in Section 1 is undecidable. However, it becomes more manageable if certain conditions hold, as3.4 and 3.5 show.

3.1. A simple agent framework

A majority of agent research focuses on communities of agents [8] and [10]. Thus, many elaborate frameworks for agent interactions have been proposed in the literature[4] and [15]. The framework we use in this paper, however, is very simple. We envision a group of agents communicating with one another through an array of resources that are necessary for the agents to carry out their jobs. Each register in this array corresponds to a single type of resource. (Because of this one-to-one correspondence, the terms “registers” and “resources” are used interchangeably from here on.) Some registers correspond to real-world resources such as money, vehicles, and data files. Some registers, however, are merely for the internal workings and communications of the agents and are of no interest to the real world. In this sense, these registers only contain control data. Nevertheless, we use the generic term “resource” to mean both types. Each agent consumes some resources in this array and produces some resources back to this array as it carries out its job. Array access is strictly sequential. That is, only a single agent has access to this array at any given moment and all other agents cannot do their jobs. Schematically, this framework is shown in Fig. 4.

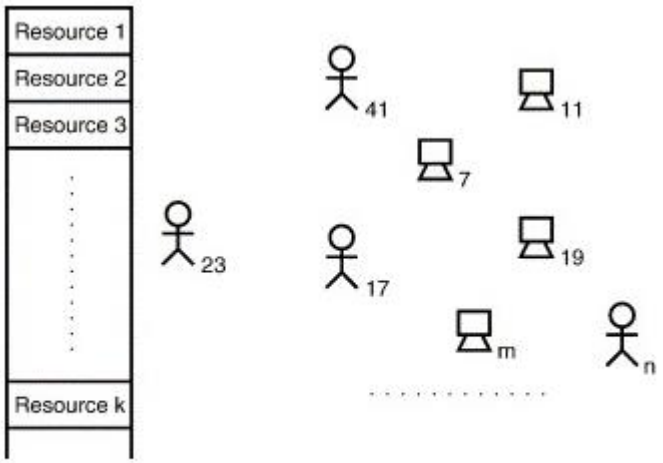

Fig. 4.
A simple agent framework.

At this moment, we assume that Agent 23 (which happens to be a human) has sole access to this array of resources. Agent access to the array depends on the actual workflow. The workflow then determines the order of access for the agents. We also assume that the array encompasses the whole environment of interest for all agents. That is, all stimuli that an agent may perceive are stored in the array as control data.

In this framework, an agent cannot carry out its job without the resources in the array. Since only a single agent can access the array at any given time, no two agents can do their jobs simultaneously. Thus, this framework does not support parallelism. By avoiding parallelism,

deadlocks, concurrency controls and other problems associated with parallelism are no longer an issue [1]. This limitation is reasonable since we are studying the capability of a group of agents, not parallelism. In fact, disallowing parallelism creates no loss of generality as far as deciding the capability of a group of agents is concerned, as we shall formally prove in Section 3.2.

Note that we do not claim our agent framework is superior to other agent frameworks, nor do we claim abacus programs are appropriate for modeling workflows. In fact, we have already stated in Section 1 that our agent framework only allows a narrow class of workflows and admittedly, many real-world workflows do not fit in this class. Our main purpose, however, is to define an agent framework such that it is general enough to model abacus programs, and yet it is simple enough to be subsumed by other agent frameworks and thus the computational problems of our agent framework can be carried over to other agent frameworks (see Section 4.2).

3.2. Statechart modeling of our agent framework

To prove that the research question in Section 1 is undecidable in general, we need to make several assumptions. These assumptions ensure that every piece of information about the agents, the environment in which they reside, and the workflow among them can be manipulated by computers. By so doing, we show that some workflow problems are inherent in agent-based workflows, and are not caused by something that cannot be computed.

Assumption 1.

Because computers are discrete devices, computers can only manipulate discrete values. Hence, only countable resources are considered. To represent the available resources, a register in the array corresponds to a unique type of resource and it stores a nonnegative integer x ($x \geq 0$) where x means there are x units of the type of resource to which the register corresponds available to the agents. In addition, because the array encompasses the whole environment of interest to all agents, all stimuli of the environment are assumed to be encoded as nonnegative integers in the array.

Assumption 2.

An agent executes a well-defined algorithm in carrying out its job if the necessary input resources are available. That is, it executes an algorithm in transforming the required input resources into the intended output resources. In contrast, if the input resources are not available, an agent will not do its job. Since an agent executes an algorithm in carrying out its job, it always terminates its job in a finite number of steps.

Now, we can formally define what an agent is for this research. An *agent A* is a transformation that transforms certain input resources into certain output resources. Consistent with our second assumption, such a transformation is algorithmic.

$$A(n_1 r_1 \ldots, n_p r_p) = (n_{p+1} r_{p+1}, \ldots, n_q r_q), 0 \leq p \leq q \quad (1)$$

and n_i§$(1 \leq i \leq q)$ are all positive integer constants.

This transformation means that agent *A* consumes n_1 units of resource $r_1, \ldots,$ *np* units of resource *rp* and produces np_{+1} units of resource $rp_{+1}, \ldots,$ *nq* units of resource *rq*. After an agent has successfully completed its transformation, it signals "success" to its environment. On the contrary, if an agent's input resources are not available, it signals "failure" to its environment. Note that the sets of input resources and output resources of an agent are not necessarily disjoint. Some resources may appear in both sets. In such a case, we can infer that those resources are used by the agent in performing its job and are returned when the job is complete.

Example 1.

A sample machinist, a cutter, a sewing machinist, a clothing presser, and a garment examiner are responsible for the workflow in Fig. 1, one for each step of the process. Written as functions, they are represented as follows:

> *Sample Machinist* (*a garment pattern*) = (*a garment pattern, a sample garment*),
> *Cutter* (*a garment pattern*) = (*a garment pattern, a collection of pieces*),
> *Sewing Machinist* (*a collection of pieces*) = (*a wrinkle and unchecked garment*),
> *Clothing Presser* (*a wrinkle and unchecked garment*) = (*a wrinkle free and unchecked garment*),
> *Garment Examiner* (*a wrinkle free and unchecked garment*) = (*a wrinkle free and checked garment*).

In terms of statecharts, we model the array of resources as an array of nonnegative integers and an agent as a state that executes the algorithm for the transformation of the agent, as shown in Fig. 5. The two outgoing transitions in Fig. 5 point to the next agents that have access to the array. Effectively, this defines a workflow among the agents. Formally, *a workflow of agents in this framework* is a statechart in which each state corresponds to an agent and each executes an algorithm that transforms certain input resources into certain output resources. Each state has zero, one, or two outgoing transitions that point to the next agents that have access to the array of resources. If the transformation is successfully performed, the workflow follows the transition that checks for the condition [Flag = Success] (abbreviated as [S]) to locate the next agent. If not, the transition that checks for the condition [Flag = Failure] (abbreviated as [F]) is followed. One of the states in the workflow is defined to be the start state, which is the first one to be executed. As an example, a particular workflow of the agents in Fig. 4 is shown in Fig. 6. Note that the state that corresponds to Agent 17 has no [F] outgoing transition and the one for Agent *m* has no [S] outgoing transition. Finally, the one for Agent *n* has no outgoing transition whatsoever.

Theorem 2.

Workflows of agents in our agent framework are capable of modeling abacus programs.

Proof.

To prove this theorem, it suffices to show that given an abacus program *P*, we can find a workflow of agents that computes precisely the same set of functions. The infinite array of nonnegative integers is immediately available to the workflow. For each node of addition

reg(i) + in P, we define an agent Ai_+ whose transformation is $Ai_+() = (1i)$ by setting $p = 0$ and $q = 1$ in Eq. (1). Ai_+ consumes nothing but produces one unit of the resource stored in register i. Since this transformation always succeeds, Ai_+ always signals success. For each node of subtraction reg(i)− in P, we define an agent Ai_- whose transformation is $Ai_-(1i) = ()$ by setting $p = q = 1$ in Eq. (1). $Ai-$ produces nothing but consumes one unit of the resource stored in register i if there is at least one unit there to begin with. If reg(i) > 0, then this transformation can be performed and Ai_- signals success. However, if reg(i) = 0, then Ai_- cannot subtract one from register i. In this case, Ai_- signals failure.

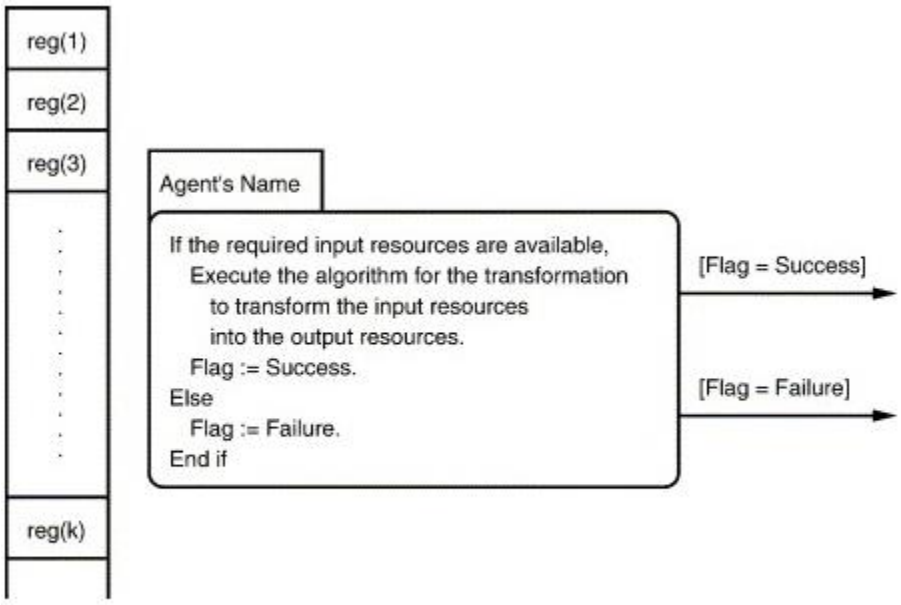

Fig. 5.
A statechart model of an agent.

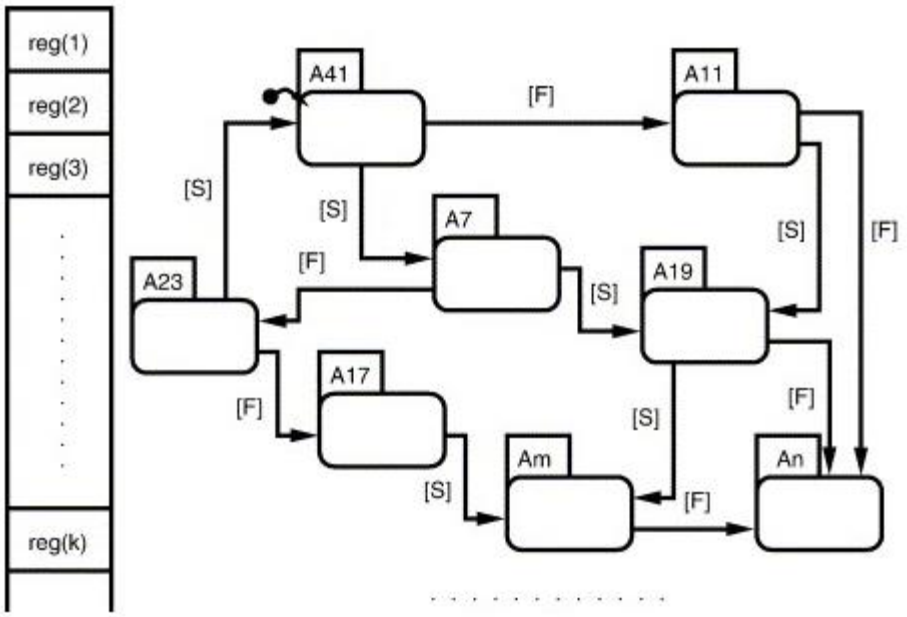

Fig. 6.
A workflow of agents in our framework.

Since abacus programs and Turing machines have the same modeling power (see Chapter 7 of [12]), by Theorem 2, workflows of agents in our framework and Turing machines also have the same modeling power. Thus, even though our framework is quite primitive, there is no loss of generality.

3.3. Computational problems of agent-based workflows

Now we are ready to prove Theorem 3 and Theorem 4, which collectively provide a negative answer to the research question in Section 1. Theorem 3 proves that if someone gives us a workflow of agents, it is undecidable to determine whether such a workflow is able to produce a predetermined number of units of a quantifiable resource. Note that for Theorem 3, the workflow of agents has already been specified. On the contrary, Theorem 4 proves that if someone gives us a group of agents without specifying any workflow on them, it is still undecidable to determine whether such a group of agents is able to produce a predetermined number of units of a quantifiable resource. That is, for Theorem 3, the workflow is specified but for Theorem 4, the workflow is unspecified. In this sense, Theorem is stronger than Theorem 3.

Theorem 3.

Whether a workflow of agents can obtain a certain number of units of a quantifiable resource is undecidable.

Proof.

By Assumption 1, every resource of interest is quantifiable. By Assumption 2, every agent executes an algorithm to carry out its job. Therefore, a workflow of agents is a statechart model of a well-defined algorithm in which each agent can be viewed as a subroutine in the overall

algorithm. Since a workflow of agents is a statechart, we may add a high-level state over it. At the same time, we may also add a transition from this high-level state to a state named "done". This transition is further guarded by a nontrivial condition [*x* units of resource *y* are produced], as Fig. 7 shows. Since this transition originates from a high-level state, it has a higher priority than all the other transitions in the statechart [7]. Therefore, when this condition becomes true, every state in the statechart will be exited immediately and the done state will be entered at once. Hence, the goal "*x* units of resource *y* are produced" is achieved if and only if the done state is entered. The undecidability of this theorem thus follows from Corollary 1.

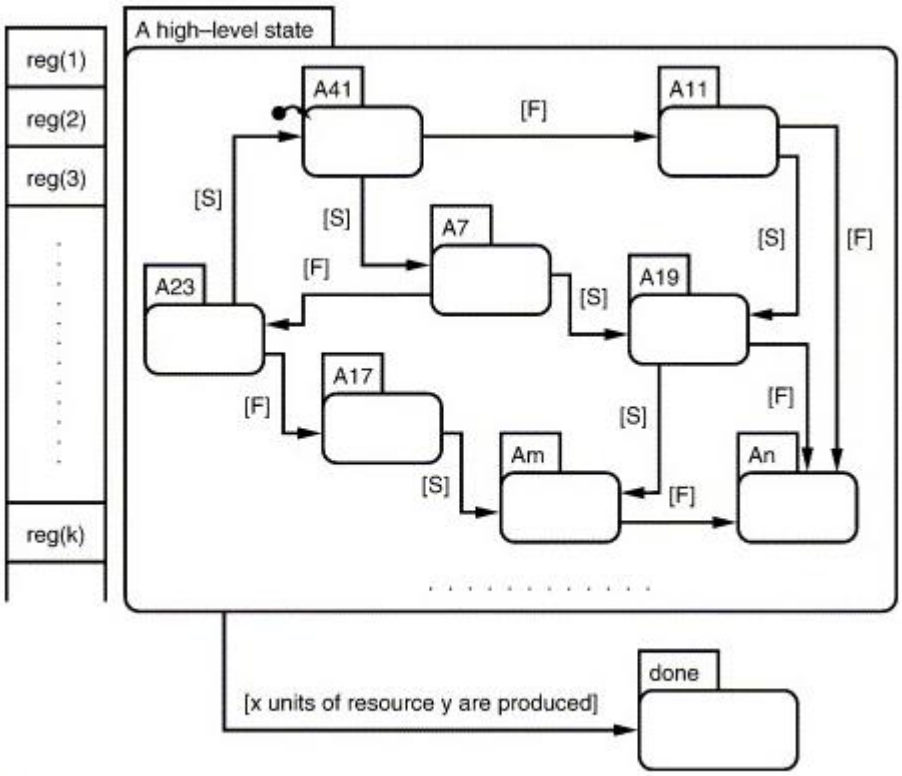

Fig. 7.
The done state of a workflow of agents.

Corollary 3.

Determining the maximum number of units of a quantifiable resource produced by a workflow of agents is undecidable.

Proof.
If there is such an algorithm, it can be used to solve the problem of Theorem 3, which is a contradiction.
Theorem 4.

Whether a set of agents can obtain a certain number of units of a quantifiable resource is undecidable.

Proof.

We first assert that the number of possible workflows of a set of agents is finite. Suppose that there are n agents. Because every agent is represented by a state, there are n states. Let A be an agent and SA be the state that represents A. Since each state is allowed to have zero, one, or two outgoing transitions that point to the next agents, we have the following cases to consider for SA:

1. SA has no outgoing transition: There is only one possibility.

2. SA has the [S] outgoing transition only: There are n possibilities since any one of the$n$ states, including SA itself, can be the target state.

3. SA has the [F] outgoing transition only: There are n possibilities since any one of the$n$ states, including SA itself, can be the target state.

4. SA has both [S] and [F] outgoing transitions: There are n^2 possibilities, which is the product of the two previous cases.

Thus, totally there are $n^2 + 2n + 1 = (n + 1)^2$ possibilities for SA being the source state. Since there are n states, the number of possibilities is $(n + 1)^2n$. However, any one of the$n$ states can be the start state. Therefore, the total number of workflows that can be constructed from n agents is $n(n + 1)^2n$ — a finite number. One may suggest executing these $n(n + 1)^2n$ workflows in parallel to search for a workflow that produces the predetermined number of units of a quantifiable resource for these n agents. By Theorem 3, this search is impossible since some of these workflows may never halt and thus the search may go on forever.

3.4. Bounded registers and computational problems

If every referenced register has an upper limit during the course of execution of a workflow of agents, the problems presented in Section 3.3 become decidable. That is, computers can solve the workflow problems in Section 3.3 if for every referenced register reg(i), $0 \leq \text{reg}(i) < k$ for some k. To proceed, we first discuss three types of statechart computations that occur within our framework.

1. A sequence of stages in the computation keeps occurring repeatedly.

2. Every stage occurring in the computation is different. However, the number of different stages of the computation is infinite.

3. Every stage occurring in the computation is different. However, the number of different stages of the computation is finite.

A computation of the first two types goes on forever. A computation of the third type, however, halts after a finite number of stages. Computations of the second type are in fact the main cause of the computational problems in Section 3.3.

Example 2.

Depending on the initial values of reg(1) and reg(2), the workflow of the five software agents in Fig. 8 exhibits all three types of computations, as shown in Fig. 9. We illustrate them in turn. Suppose reg(1) = 8 and reg(2) = 6. That is, the initial stage is [8, 6, A_1]. According to the workflow, two stages [8, 6, A_1] and [6, 8, A_2] keep occurring repeatedly. Thus, this computation never halts, as the first column in Fig. 9 shows. On the contrary, if the initial stage is [6, 5, A_1], the workflow alternates itself over state A_3 and state A_4forever. A_3 adds 2 to reg(2) and A_4 adds 1 to reg(1). Thus, the stages of this computation are all different since the numbers in the two registers are all different, as the second column in Fig. 9 shows. Lastly, if the initial stage is [5, 0, A_1], the workflow goes through 5 stages and halts. When it halts, the workflow halts at state A_5, as the third column in Fig. 9shows.

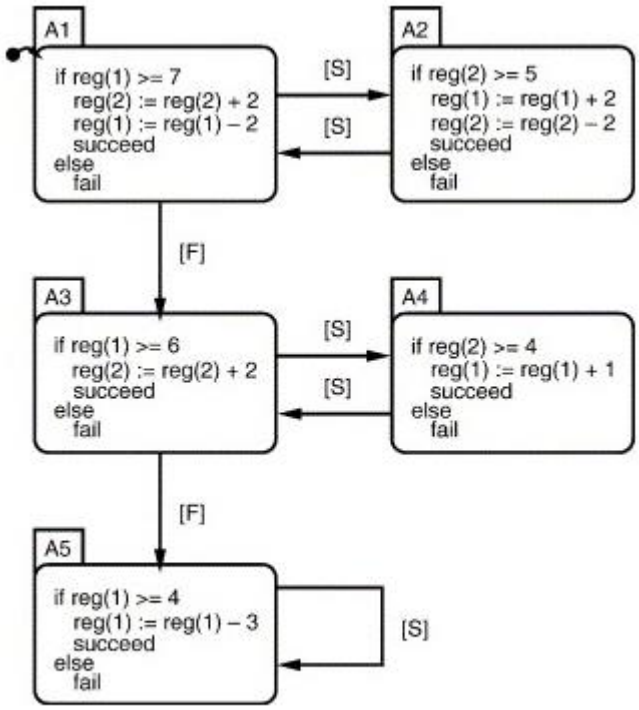

Fig. 8.
A workflow of software agents.

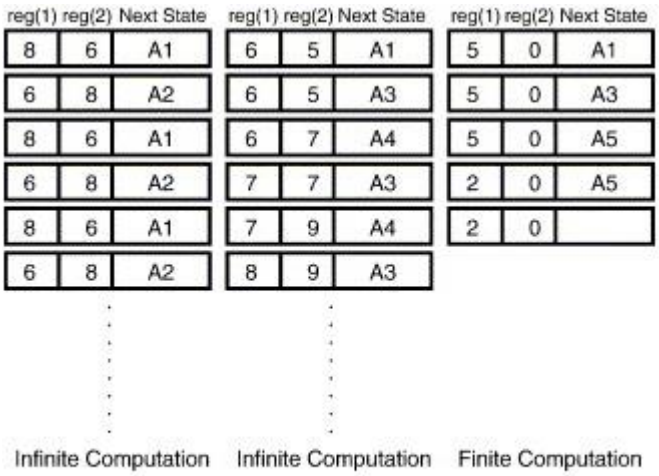

Fig. 9.
Three types of statechart computations.

Theorem 5 demonstrates that the problem of Theorem 3 becomes decidable if every referenced register is bounded by a number. Essentially, if every referenced register is bounded by a number, a second type computation cannot happen. Since all of the different stages in a first type or a third type computation can be enumerated in a finite number of steps, the problem of Theorem 3 becomes decidable, as we shall demonstrate. Similarly, the problems of Corollary 3 and Theorem 4 become decidable if every referenced register is bounded by a number.

Theorem 5.

Whether a workflow of agents can obtain a certain number of units of a quantifiable resource is decidable if every referenced register is bounded by a number.[3]

Proof.

Given a workflow *W* of agents, suppose that *W* has *m* states and *W* references *n*registers. Let *k* be a bound of these *n* registers. That is, $0 \leq \text{reg}(i) < k$ for $1 \leq i \leq n$. Therefore, the number of different stages that *W* goes through is bounded by the number$m(kn)$. Assume that the problem in the theorem statement of Theorem 3 is concerned with producing *x* units of a quantifiable resource *y*. That is, we are interested in knowing whether *W* will reach a stage in which $\text{reg}(y) \geq x$. We now describe an algorithm that searches for such a stage. If such a stage is found, the answer is “Yes”; otherwise, the answer is “No”.
1. Set up all referenced registers according to the resources that are initially available to *W* and set the Next State field to the initial state of *W*.

2. Execute the algorithm of the state specified in the Next State field and reset the Next State field according to *W*.

3. If *W* reaches a stage that has occurred before, stop immediately.

4. Go to Step 2 if the Next State field is not empty. Otherwise, stop immediately.

At Step 3, since a workflow of agents is sequential and deterministic, if W reaches a stage that has occurred before, W enters an infinite loop. Like the first column in Fig. 9, we have successfully enumerated all the different stages that W goes through. Therefore, we may stop. If W never reaches a stage that has occurred before, since the number of different stages that W goes through is bounded by the number $m(kn)$, eventually W will exhaust all of them. At that point in time, the Next State field becomes empty and all the different stages that W goes through have been successfully enumerated, like the third column in Fig. 9. If $reg(y)$ is greater than or equal to x in any of the stages enumerated by this algorithm, the answer is "Yes". Otherwise, the answer is "No". In fact, we may stop earlier if W reaches a stage such that $reg(y) \geq x$, without having to enumerate all the different stages that W goes through.

Corollary 4.

Determining the maximum number of units of a quantifiable resource produced by a workflow of agents is decidable if every referenced register is bounded by a number.

Proof.

We use the same searching algorithm for the proof of Theorem 5 here. If every referenced register is bounded by a number, we can enumerate all the different stages that the workflow goes through, as shown in the proof for Theorem 5. Thus, we can examine each stage in turn to find out the maximum number.

Corollary 5.

Whether a set of agents can obtain a certain number of units of a quantifiable resource is decidable if every referenced register is bounded by a number.

Proof.

As noted in the proof for Theorem 4, the number of workflows of a set of agents is finite. This corollary then follows immediately from Corollary 4. □

Even though the condition that no unbounded register is sufficient to guarantee decidability, such a condition itself is undecidable, as Theorem 6 shows. Example 3 is a real-life example that shows setting up bounds are indeed impossible for some situations.

Theorem 6.

Whether a register is bounded or not in a statechart is undecidable.

Proof.

We use a similar construction proof to the one for Theorem 3 here. When the condition "$x$units of resource y are produced" becomes true, the statechart enters a state that keeps incrementing register k that is not referenced anywhere in the statechart, as Fig. 10shows. Since $reg(k)$ is not referenced in the statechart, initially $reg(k) = 0$. Therefore, the condition "x units of

resource *y* are produced" is false if and only if register *k* is bounded by the number 1. The undecidability of this theorem thus follows from Corollary 2.

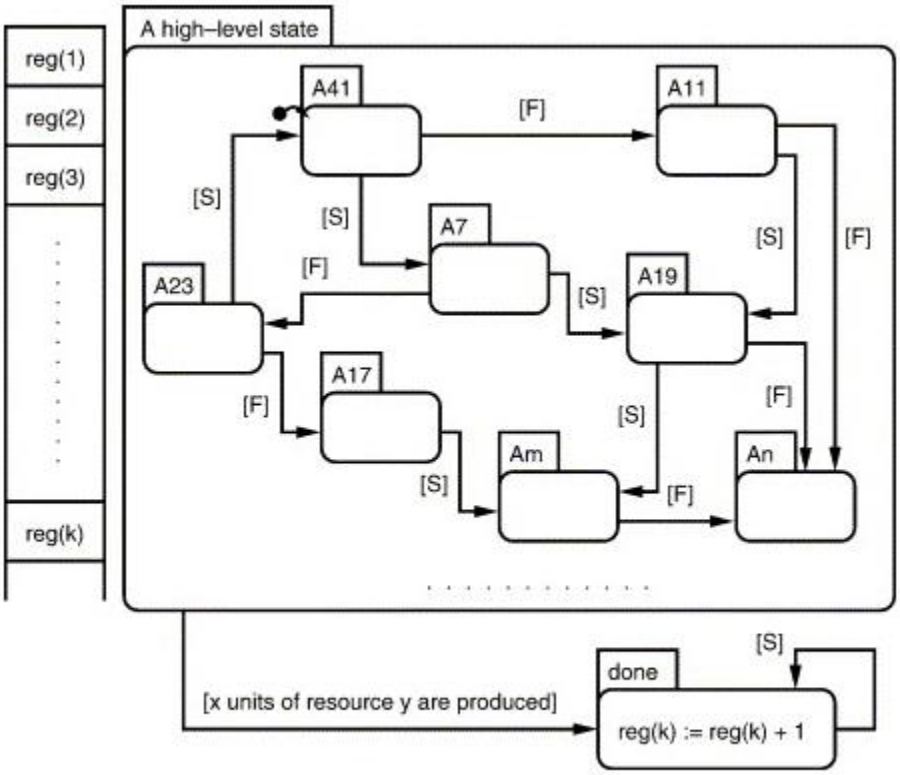

Fig. 10.
An unbounded register.

Example 3.

Fig. 11 shows a set of software agents which work together for a common objective. The interface agents IA_1, IA_2, …, IAn ($n \geq 1$) collectively function as an operating system, which check for the calling conditions of the software agents SA_1, SA_2, …, SAn ($n \geq 1$). If the calling conditions are met, they then call on the software agents to execute. When the software agents are done, they pass the control back to the interface agents. By the Rice's Theorem [16], software verification is undecidable. Thus, setting up bounds for the agents in Fig. 11 is impossible; otherwise, software verification becomes decidable—a contradiction.

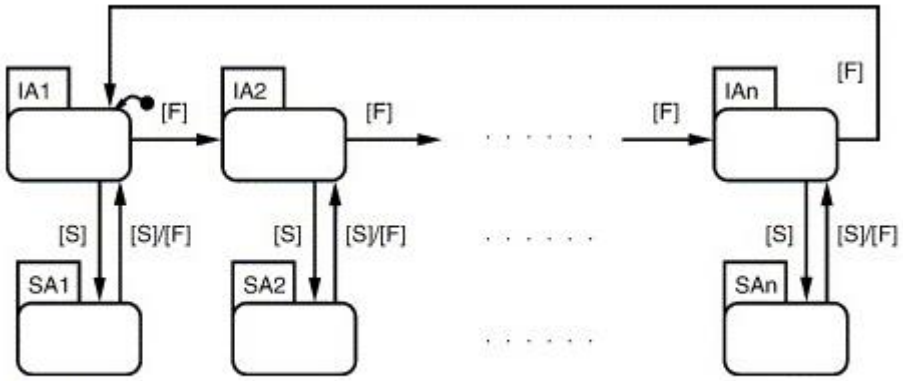

Fig. 11.
A set of software agents.

3.5. Limited number of executions

Theorem 5 and its corollaries state that if every referenced register is bounded by a number, then the problems in Section 3.3 become decidable. On the contrary, Theorem 6states that determining if a register is bounded or not in a workflow of agents is not decidable. Therefore, some other means other than computers must set a bound on each register. For example, a company's policy might dictate that such a bound must exist for each resource in the course of a workflow execution. Such a policy, however, is cumbersome. Lemma 1 below shows a more intuitive condition that guarantees every referenced register is bounded by a number.

Lemma 1.

If there is a bound on the number of times an agent can execute in a workflow, then every referenced register must be bounded by a number.

Proof.

By Assumption 2, each agent terminates its job in a finite number of steps. Therefore, the number of units an agent can contribute to a resource is limited in a single execution of its algorithm. Thus, if there is a bound on the number of times an agent can execute in a workflow, there is a bound on each referenced register.

Corollary 6.

Every referenced register is bounded by a number if a workflow of agents does not have a loop.

Proof.

If a workflow of agents does not have a loop, every agent can at most execute once. This corollary thus follows from Lemma 1.

4. Related work

In this section, we discuss four types of related work. First, we prove that our agent framework subsumes Petri Nets [12], which are a very popular modeling language. Second, we demonstrate that the multi-agent framework in [15] subsumes our agent framework as a special case and thus the computational problems in Section 3.3 apply to the framework in [15] as well. Third, we prove that if the number of times an agent can execute in a workflow is bounded, and no agent consumes any resource produced by any agent (including itself), then the problem of Theorem 4 can be solved by linear programming. On the other hand, if there are some resources that are both produced and consumed by some agents, then we show by a counter-example that linear programming cannot solve the problem of Theorem 4. Fourth, we address the issue of workflow verification.

4.1. Petri Nets and our agent framework

Petri Nets [12] as a modeling language has been used in the development of many manufacturing systems. Here we show that our agent framework subsumes Petri Nets, but Petri Nets do not subsume our agent framework. In other words, every system that can be modeled by Petri Nets can also be modeled by our agent framework. However, there are some systems that can be modeled by our agent framework but not by Petri Nets. Our argument relies on many proven results in the literature and they are listed as follows.

1. Petri Nets are nondeterministic (see page 36 in [12]). That is, Petri Nets are inherently parallel.

2. Petri Nets can be modeled by nondeterministic Turing machines (see Chapter 7 in[12]).

3. Nondeterministic Turing machines can be modeled by deterministic Turing machines (see Chapter 3 in [16]).

4. Deterministic Turing machines can be modeled by abacus programs (see Chapters 6, 7, and 8 in [3]).

5. Abacus programs can be modeled by workflows of agents (see Theorem 2).

The limited modeling power of Petri Nets is due to the fact that Petri Nets cannot test an unbounded place for zero (see page 68 in [12]). On the other hand, our agents can easily test a register, which is unbounded, for zero and thus Petri Nets cannot model our agent framework. Thus, it suffices to demonstrate that there is a system that can be modeled by our agent framework but not by Petri Nets. The readers/writers problem is such a system (see page 66 in [12]).

4.2. The multi-agent framework in [15]

Since our agent framework is very simple, if the capability of a group of agents is undecidable in our framework, such a problem remains undecidable in other multi-agent frameworks that subsume ours as a special case. To make this notion concrete, in this section we show specifically how the multi-agent framework in [15] subsumes ours. Since the number of multi-agent frameworks is unlimited, we cannot possibly elaborate on every single one of them.

However, by proving our agent framework is a special case of the framework in [15], the reader should be convinced that our agent framework can be easily subsumed in other multi-agent frameworks.

We first provide some background information of the framework in [15]. The authors of [15] define a multi-agent framework for the coordination and integration of information systems. The underlying idea is that an information system is considered to be a collection of stand-alone units, which can be viewed as agents. Thus, to integrate multiple information systems is to coordinate the agents in these different information systems in such a way that the overall business goal can be achieved. For this purpose, the authors define a multi-agent framework to analyze and design such an integrated system. To demonstrate the usefulness of their framework, the authors apply their framework to the development of a manufacturing information system for managing the production processes for making printed circuits boards.

To simplify the presentation, we let A denote the array of nonnegative integers and let $\bar{A}_t$ denote the content of A at time *t*. We now show point-by-point that how the multi-agent framework in [15] subsumes ours as a special case. In other words, we show that every single detail of our agent framework can be modeled in the framework of [15]. We first consider an individual agent, and then we consider the system as a whole. For each individual agent *Ai*, the following points are sufficient to model *Ai*.

1. *Ai*'s local state at time *t* is $\bar{A}_t$. In the notation of [15], *sit* at time *t* is $\bar{A}_t$ and *Si* is the set of all possible *sit*'s.

2. *Ai*'s protocol is the algorithm that defines the behavior of *Ai* (see Assumption 2 in Section 3.2).

3. There are two possible activities for *Ai* and which one to choose depends on *Ai*'s protocol and *Ai*'s local state at the time *Ai* is called. (1) If *Ai*'s input resources are available, then *Ai* consumes the input resources and produces the output resources, as defined by Eq. (1). Afterward, *Ai* locates the next agent to be called in the workflow by following the [S] outgoing transition (if there is one). (2) If *Ai*'s input resources are *not* available, then *Ai* locates the next agent to be called in the workflow by following the [F] outgoing transition (if there is one).

4. *Ai*'s acquaintances are exactly the agents pointed at by *Ai*'s [S] outgoing transition (if there is one) and *Ai*'s [F] outgoing transition (if there is one). That is, in the notation of [15], $\gamma[i, j] = 1$ if and only if *Ai*'s [S] outgoing transition or *Ai*'s [F] outgoing transition leads to agent *Aj*.

For the system as a whole, the following points suffice to model it.

1. The environment *E* in [15] is A in this case since we assume A encompasses the whole environment of interest for all agents, as stated in Section 3.1.

2. The global state of the system at time *t* is $\bar{A}_t$ plus the next agent *Ai* to be called in the workflow (if there is one). That is, in the notation of [15], *gt* at time *t* is a pair ($\bar{A}_t$, *Ai*). The global structure **G** is the set of all possible *gt*'s.

3. The performance measure of the system at time t is the content of the register that corresponds to the desired output resource at time t. That is, in the notation of [15],$\pi(gt)$ is reg(k) where register k represents the desired output resource.

4. The joint protocol of the system at time t is the protocol of the active agent at time t. (There is only one active agent at any point in time since our agent framework is strictly sequential.)

5. The joint activity of the system at time t is the activity of the active agent at time t.

6. Temporal interdependency among agents is determined entirely by the workflow.

7. Resource interdependency does not apply since access to A is strictly sequential and thus no two different agents have access to A at the same time. Thus, no coordination of resource allocation is required.

8. Sub-goal interdependency does not apply since an agent's behavior is determined entirely by its Eq. (1), which simply consumes certain input resources (if they are available) and produces certain output resources without regards to any other agents.

9. Our agent framework applies decentralized control and it is entirely determined by the workflow.

We conclude that the multi-agent framework in [15] subsumes our agent framework in an obvious way.

4.3. Linear programming and our agent framework

Here we prove that a certain agent-based workflow problem can be solved by linear programming—a common operations research technique [9].

Corollary 7.

If there is a bound on the number of times an agent can execute in a workflow, and no agent consumes any resource produced by any agent (including itself), then the problem of Theorem 4 can be solved by linear programming.

Proof.

Since no agent consumes any resource produced by any agent, all agents can execute independently, which means they can execute in any order. By Eq. (1), each agent consumes constant amount of input resources and produces constant amount of output resources in a single execution. Moreover, because the number of times an agent can execute is bounded, the maximum number of times an agent can execute then depends solely on the initial amount of resources available to the workflow. Further, resources initially available to the workflow cannot

be replenished. This means that a set of linear inequalities can be used to determine the correct number of times an agent should execute in order to solve the problem of Theorem 4.

Example 4.

Consider two agents A_1 and A_2 whose equations are $A_1(1r_1, 2r_2) = (1r_3)$ and $A_2(2r_1, 1r_2) = (1r_3)$. Since A_1 does not consume any resource produced by A_2 and vice versa, $A_1$and A_2 can execute independently. Suppose there are 7 units of r_1 and 6 units of $r_2$initially available to A_1 and A_2 and the goal is to maximize the production of r_3. Let x_1 andx_2 be the number of times A_1 and A_2 should execute. This problem can be expressed in linear programming as follows.

Maximize $x_1 + x_2$ subject to these inequalities:

$$\begin{aligned} &x_1 \geq 0, \\ &x_2 \geq 0, \\ &x_1 + 2x_2 \leq 7, \\ &2x_1 + x_2 \leq 6. \end{aligned}$$

The problem can then be solved with standard operations research methods and the integer solution is $x_1 = 2$ and $x_2 = 2$.

Example 5 shows that if the condition of Corollary 7 does not hold, linear programming cannot solve the problem of Theorem 4.

Example 5.

Consider two agents A_1 and A_2 whose equations are $A_1(1r_1, 1r_2) = (1r_3)$ and $A_2(1r_1, 1r_3) = (1r_2, 1r4)$. In a single execution, A_1 produces 1 unit of r_3, A_2 consumes 1 unit of r_3,A_2 produces 1 unit of r_2 and A_1 consumes 1 unit of r_2. That is, the condition of Corollary 7 does not hold. Suppose that the goal is to produce 1 unit of r_4 and we want to find out the least amount of initial resources we can provide to these two agents. Let x_1 and x_2 be the number of times A_1 and A_2 should execute in a workflow. x_1 and x_2 are related by the following linear inequities:

$$\begin{aligned} &x_1 \geq 0, \\ &x_2 \geq 0, \\ &x_1 + x_2 \leq a, \\ &x_1 - x_2 \leq b, \\ &x_2 - x_1 \leq c, \\ &x_2 \geq 1. \end{aligned}$$

The variables a, b, and c denote the initial numbers of units of r_1, r_2, and r_3 provided to these agents. In an attempt to minimize a, b and c as much as possible and since $x_2 \geq 1$, one may set x_1 to 1 and set both b and c to 0. However, if $b = c = 0$, neither A_1 nor A_2 can be activated and

thus these inequalities lead to a mistaken solution. Unlike Example 4where the agents can execute independently of each other, A_1 and A_2 of this example are interdependent because each produces some resources needed by the other and thus the order of execution of A_1 and A_2 becomes important. However, the order of execution of A_1 and A_2 cannot be captured by linear inequalities.

4.4. Workflow verification

Validation, verification, and performance analysis are three main types of analysis for workflows [18]. Validation tests a workflow with some known test cases to see if it behaves as expected. Test cases, however, can only reveal existing problems but cannot prove the correctness of a workflow. Performance analysis evaluates a workflow to see if it meets the predefined requirements with respect to throughput times, service levels, and resource utilization. This research, however, addresses a more fundamental problem. We would like to know if a group of agents is able to achieve a business goal defined in terms of acquiring a specific number of units of a quantifiable resource. To us, a workflow of agents is correct if it can produce the required number of units of a specific resource. Performance analysis is pointless if a workflow of agents is incorrect. Since verification is concerned with the correctness of a workflow, verification is our main focus in this section.

Some of the notable researches on workflow verification are [14], [17] and [18]. In [18], Task Structures are first mapped into Workflow Nets, which are a subclass of ordinary Petri Nets [12]. Analysis is then done on Workflow Nets instead. The authors provide a necessary and sufficient condition for a Workflow Net to be sound. However, a sound Workflow Net, as defined in [18], does not necessarily mean it is correct in our sense and vice versa. In other words, it is possible for a sound Workflow Net not to be able to produce a specific number of units of a quantifiable resource.[4] Equally possible is that a workflow of agents might not be sound but be able to produce the same business goal.[5]In addition, ordinary Petri Nets cannot test an unbounded place for zero; while it is quite difficult to imagine a human agent cannot do the same. Hence, the results in [18] are independent with those in this research. Nevertheless, sound Workflow Nets always terminate. As a result, sound Workflow Nets do not have the computational problems of this research. In short, [18] is more concerned with the liveness and boundedness of Workflow Nets. This research, on the contrary, focuses on whether or not a business goal can be achieved by a group of agents.

5. Conclusions

This paper rigorously proves that there is no computer-based decision support system that is able to solve the research question in Section 1 unless we restrain it in some ways. This means that less precise methods, potentially assisted by computers, must be sought to provide partial solutions to it. Since precise algorithmic solutions do not exist, approximations and guesswork must inevitably be introduced. Since computers cannot guess, human intervention becomes necessary. Other contributions of this paper include a formal proof that if the quantity of each resource of a workflow is limited, the research question in Section 1 becomes decidable. It then follows that if the number of times an agent can execute is finite, computers can solve it. However, setting a bound for the number of times an agent can execute in a workflow is a

managerial issue, which must be done by humans and cannot be automated. We also study the relationships between this work and Petri Nets, the multi-agent framework in [15], linear programming and workflow verification. We prove that our agent framework subsumes Petri Nets but the multi-agent framework in [15] subsumes ours as a special case, which means the computational problems we discovered in this paper apply to the framework in [15] as well.

Acknowledgements

We would like to thank the anonymous referees for their helpful suggestions. W. Y. Mok was supported in part by the Richard A. Witmondt Faculty Fellowship and a UAH Research Mini-Grant and he would like to thank Kit Yee Cheung for brainstorming the original idea of this research.

Appendix A.

Proof for Theorem 1.

It suffices to show that transforming an abacus program into a statechart is algorithmic. If such a transformation is algorithmic and there is an algorithm that decides whether a statechart will halt or not, the same algorithm can also decide indirectly if an abacus program will halt or not—a contradiction. Given an abacus program, the resulting statechart has access to the same infinite array of registers. Each node labeled with reg(i)+ is transformed into a state that calls a subroutine add(i) which performs the intended addition. Likewise, each node labeled with reg(i)− is transformed into a state that calls a subroutine sub(i) which performs the intended subtraction. It is clear that this transformation is algorithmic and thus the proof is complete.

Proof for Corollary 1.

Consider an abacus program P in P. Since P is finite, the number of registers referenced in P is finite. Since an abacus has an unlimited number of registers, there is a register, say register x, that is not referenced in P. Create a new node N with the label reg(x)+ and without any outgoing arrow. Supply all missing outgoing arrows in P so that they all point at N. Call this resulting abacus program P'. Since register x is not referenced in P and N has no outgoing arrow, P and P' compute exactly the same functions. Thus, P halts if and only if P' halts at N. Now transform P' into a statechart S using the transformation process in the Proof for Theorem 1. Let H be the state in S that corresponds to N in P'. Thus, S enters H if and only if P' halts at N. If there exists an algorithm A that can decide if a particular state will be entered in a statechart, A can decide if S will enter H. Since all these transformations are algorithmic and S enters H if and only if P halts, A can decide if P will halt. Since P is chosen arbitrarily from P, A can solve the halting problem of abacus programs — a contradiction.

Proof for Corollary 2.

The same construction proof for Corollary 1 is used here. The nontrivial condition is in (H) however. in (H) is a predicate that returns true if statechart S is in state H and returns false

otherwise [7]. Thus, *P* halts if and only if in (*H*) returns true and we obtain a similar contradiction to the one in the Proof for Corollary 1.

References

[1] P.A. Bernstein, N. Goodman. **Concurrency control in distributed database systems.** Computing Surveys, 13 (2) (1981 (June))

[2] G. Booch, J. Rumbaugh, I. Jacobson. **The Unified Modeling Language: User Guide.** Addison-Wesley, Reading, Massachusetts (1999)

[3] G.S. Boolos, R.C. Jeffrey. **Computability and Logic.** (third edition) Cambridge University Press, Melbourne, Australia (1989)

[4] M.H. Chang, J.E. Harrington. **Centralization vs. decentralization in a multi-unit organization: a computational model of a retail chain as a multi-agent adaptive system.** Management Science, 46 (11) (2000 (November)), pp. 1427–1440

[5] T.H. Davenport, J.E. Short. **The new industrial engineering: information technology and business process redesign**. Sloan Management Review, 31 (4) (1990 (Summer)), pp. 11–27

[6] M. Hammer. **Reengineering work: don't automate, obliterate.** Harvard Business Review (1990 (July/August)), pp. 104–112

[7] D. Harel, A. Naamad. **The statemate semantics of statecharts.** ACM Transactions on Software Engineering and Methodology, 5 (4) (1996 (October)), pp. 293–333

[8] C.C. Hayes. **Agents in a nutshell—a very brief introduction**. IEEE Transactions on Knowledge and Data Engineering, 11 (1) (1999 (January/February)), pp. 127–132

[9] F.S. Hillier, G.J. Lieberman. **Introduction to Operations Research.** (seventh edition)McGraw Hill, Boston (2001)

[10] V.R. Lesser. **Cooperative multiagent systems: a personal view of the state of the art**. IEEE Transactions on Knowledge and Data Engineering, 11 (1) (1999 (January/February)), pp. 133–142

[11] W.Y. Mok, D. Paper. **Using Harel's statecharts to model business workflows.** Journal of Database Management, 13 (3) (2002 (July–Sept.)), pp. 17–34

[12] J.L. Peterson. **Petri Net Theory and The Modeling of Systems.** Prentice Hall, Englewood Cliffs, New Jersey (1981)

[13] S.J. Russell, P. Norvig. **Artificial Intelligence: A Modern Approach.** (second edition)Prentice Hall, Upper Saddle River, New Jersey (2003)

[14] W. Sadiq, M.E. Orlowska. **Analyzing process models using graph reduction techniques.** Information Systems, 25 (2) (2000), pp. 117–134

[15] R. Sikora, M.J. Shaw. **A multi-agent framework for the coordination and integration of information systems**. Management Science, 44 (11) (1998 (November)), pp. 65–78 (Part 2 of 2)

[16] M. Sipser. **Introduction to the Theory of Computation.** PWS Publishing Company, Boston, Massachusetts (1997)

[17] A.H.M. ter Hofstede, M.E. Orlowska, J. Rajapakse. **Verification problems in conceptual workflow specifications.** Data and Knowledge Engineering, 24 (3) (1998 (January)), pp. 239–256

[18] W.M.P. van der Aalst, A.H.M. ter Hofstede. **Verification of workflow task structures: a petri-net-based approach**. Information Systems, 25 (1) (2000), pp. 43–69

[1.] *e* means empty.

[2.] If $n = 0$, P takes no input values but produces a output value.

[3.] Although in most cases the bound of a register depends on the initial values of the registers, it does not necessarily show. However, the main point is that such a bound must be determined before the computation begins and cannot be modified thereafter.

[4.] A Workflow Net with a single task is clearly sound and it is easy to find a business goal that cannot be accomplished by a single task.

[5.] We may simply add some idle agents to a correct workflow and since these idle agents never work, such a workflow is not sound in the sense of [18].