Verifying data integration agents with deduction-based models

Radosław Klimek*, Łukasz Faber* and Marek Kisiel-Dorohinicki*
*AGH University of Science and Technology
al. A. Mickiewicza 30, 30-059 Krakow, Poland
E-mail: {rklimek,faber,doroh}@agh.edu.pl

Abstract—**The paper shows how an agent-based system can be subjected to formal verification using a deductive approach. The particular system for gathering open source intelligence is considered, which is build on a framework for data integration. Techniques allowing for automatic extraction of logical specifications are described with emphasis on pattern-based and rule-based approaches. An example illustrates how the proposed method works in a scenario with iterated agent tasks combining these two approaches.**

I. Introduction

THE key concept in multi-agent systems (MAS) are intelligent interactions (coordination, cooperation, negotiation). Thus multi-agent systems are ideally suited to representing problems that have multiple problem solving methods, multiple perspectives and/or multiple problem solving entities [1]. Yet this variety of perspectives often makes the design and implementation of software MAS a really difficult task.

Formal methods enable the precise formulation of important artifacts and the elimination of ambiguity. There are two well established approaches to formal reasoning and system verification [2]. The first is based on the state exploration ("model checking") and the second is based on deductive reasoning. Model checking is an operational rather than analytic approach [3]. On the other hand, deduction-based formal verification is essential for sustainable verification leverage, characterized by intuitiveness, a top-down way of thinking, logic-based reasoning, coverage of infinite computations, etc. Temporal logic is a well established formalism which allow to describe properties of reactive systems. The semantic tableaux method, which might be descriptively called "satisfiability trees", seems intuitive and may be considered as a goal-based formal reasoning.

The contribution is based on an agent-based framework dedicated to acquiring and processing distributed, heterogeneous data collected from the various Internet sources [4]. Data processing in such a system is structuralized by means of dynamic workflows based on agents' interactions. Our goal is to provide a formal description of these interactions to make sure the system works properly. Since logical specifications are difficult to specify manually, a method for an automatic extraction of logical specifications, considered as a set of temporal logic formulae, was proposed in [5], or for a building requirements models during the software requirements elicitation in [6]. Here, a case of iterated agent tasks is considered and illustrated by a more complex scenario in a slightly different application area. The case includes both an active model and a state model for agent systems.

In the first part of the paper essential logical background is provided and the method of specification generation based on workflow describing agents' interactions is described. Then the general structure of the framework for data integration is presented. This constitutes a base for the discussion of the scenario of the particular system, which shows how the approach works in practice.

II. Logical preliminaries

Logical background which is temporal logic, semantic tableaux, and the deduction-based verification system are discussed below. *Temporal logic* TL is a formal and logical system for the specification and verification of software models and systems [7]. It introduces symbolism (unary and dual operators are $\Diamond$ for "sometime in the future" and $\Box$ for "always in the future") for representing and reasoning about the truth and falsity of formulas throughout the flow of time and taking into consideration changes of their valuation. It allows to describe both temporal relations between reached states and to specify expected properties. The attention is focused on *propositional linear-time temporal logic* PLTL, i.e. the time structure constitutes a linear and unbounded sequence. Each element in the mentioned sequence corresponds to a propositional world, i.e. *atomic propositions* AP are valued in every point of the sequence. Temporal logic and their syntax and semantics are discussed in many works, e.g. [8], [7]. Considerations in the work are limited to the *smallest temporal logic*, e.g. [9], [10], which is extension of a classical propositional calculus' to the axiom $\Box(\Psi \Rightarrow \Phi) \Rightarrow (\Box\Psi \Rightarrow \Box\Phi)$ and the inference rule $\models \Psi \Longrightarrow \models \Box\Psi$. The minimal logic, also called the K logic, is sufficient to define many system properties (liveness, safety), The following formulas may be considered as examples of this logic: $action \Rightarrow \Diamond reaction$, $\Box(send \Rightarrow \Diamond ack)$, $\Diamond live$, $\Box\neg(event)$, etc.

Semantic tableaux is a decision procedure, based on formula decomposition, for checking formula satisfiability. Even though it is known in classical logic, it can also be applied in temporal logic [11]. At each step of a well-defined procedure, some logical connectives are removed and formulas are decomposed. The method is a proof by contradiction, i.e. after negation of the initial formula, finding a contradiction in all branches means that the inference tree is *closed*, and there are no valuations that satisfy a formula. It leads to the statement that the formula before the negation is true. The method provides, through so-called *open* branches of the semantic tree, information about the source of an error, if one is found. The work [12] provides an example of the inference tree.

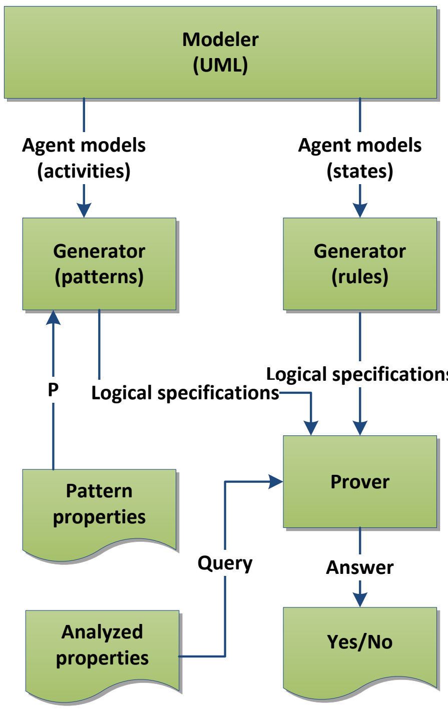

Fig. 1. A deduction-based verification system for agent models

The outline architecture of the proposed deduction-based system for agent models is presented in Fig. 1. There are two generation components. The first one works using the algorithm Γ_1 (described in the next section) and is designed for models expressed in activity diagrams using predefined workflow patterns. The second one works using the algorithm Γ_2 (described in the next section) and is designed for models expressed in state diagrams. The outputs of these generation components are logical specifications understood as sets of temporal logic formulas. The combined specification is treated as a conjunction of all formulas $p_1, \ldots, p_n = L$ and every p_i is a specification formula generated during the extraction Γ_1 or Γ_2. These formulas constitute a logical specification L. The Q formula (query) is a desired system property for the analysed software model.

Both the final specification of a system and the examined properties constitute an input to the prover component, which works using the semantic tableaux method. It enables the automated reasoning. The input for this component is the formula $C(L) \Rightarrow Q$, where $C(L)$ means conjunction of all extracted formulas, or, more precisely:

$$p_1 \wedge \ldots \wedge p_n \Rightarrow Q \qquad (1)$$

After negation of formula (1), it is placed at the root of the inference tree and decomposed using the semantic tableaux method's well-defined rules. The work [12] provides an example of the inference tree.

The whole verification procedure can be summarised in the following way:

1) Automatic generation of a logical specification based on design patterns (Γ_1);
2) Automatic generation of a logical specification based on extraction rules (Γ_2)
3) Introduction a property Q as a query for the considered model;
4) The automatic inference using semantic tableaux for the whole formula 1.

Steps 1 to 4, taken as a whole or individually, may be processed many times, whenever models are changed (step 1 or step 2) or there is a need for a new reasoning due to the revised system's property (step 3).

III. Logical specifications

Two generation methods for extraction logical specifications are considered in this section. The first one which is based on predefined workflows is discussed in a more detailed way. However, the rule-based approach is an interesting alternative for generating logical specifications.

A. A pattern-based approach

Presentation of the approach needs to introduce some basic notions and definitions. An *elementary set* of formulas over atomic formulas $a_{i,i=1,\ldots,n}$ is denoted $pat(a_i)$, or simply $pat()$, as a set of temporal logic formulas $\{f_1, \ldots, f_m\}$ such that all formulas are syntactically correct. The examples of elementary sets are $Pat1(a,b) = \{a \Rightarrow \Diamond b, \neg a \Rightarrow \neg b, \Box\neg(a \wedge b)\}$ and $Pat2(a,b,c) = \{a \Rightarrow \neg\Diamond b \wedge \Diamond c, \Box\neg(b \vee c)\}$. The logical expression enables representing nested and complex structures elementary sets. The *logical expression* W_L is a structure created using the following rules [13]:

- every elementary set $pat(a_i)$, where $i > 0$ and every a_i is an atomic formula, is a logical expression,
- every $pat(A_i)$, where $i > 0$ and every A_i is either
  - an atomic formula a_j, where $j > 0$, or
  - a set $pat(a_j)$, where $j > 0$ and a_j is an atomic formula, or
  - a logical expression $pat(A_j)$, where $j > 0$

  is also a logical expression.

The example of logical expression is $Seq(Flow(a,b,c), Switch(d,e,f))$ which is intuitive, in that it shows the sequence of a parallel split (flow) and then conditional execution (switch) of some activities.

Workflow patterns constitute a kind of primitives and enable the automation of the generation process for logical specifications. It leads to the mapping of workflow patterns to logical specifications. The proposed approach is based on the assumption that the entire activity diagrams are built using only predefined workflow patterns. The assumption is not a restriction since it enables receiving correct and well-composed systems. *Activity diagrams* of UML enable modelling workflow activities. They support choice, concurrency and iteration. The important goal of diagrams is to show how an activity depends on others [14].

```
Sequence(a1,a2):    /* ver. 13.04.2013
in={a1} / out={a2}
a1 => <>a2 / []~(a1 & a2)
SeqSeq(a1,a2,a3):
in={a1} / out={a3}
a1 => <> a2 / a2 => <> a3
[]~((a1 & a2) | (a2 & a3) | (a1 & a3))
Flow(a1,a2,a3):
in={a1} / out={a2,a3}
a1 => <>a2 & <>a3 / []~(a1 & (a2|a3))
Switch(a1,a2,a3):
in={a1} / out={a2,a3}
a1 & c(a1) => <>a2 / a1 & ~c(a2) => <>a3
[]~((a1 & a2) | (a1 & a3) | (a2 & a3))
Loop-While(a1,a2):
in={a1} / out={a1,a2}
a1 & c(a1) => <> a2 / a1 & ~c(a1) => ~<> a2
[]~(a1 & a2)
```

Fig. 2. Predefined set of pattern temporal properties

Logical properties of all design patterns are expressed in temporal logic formulas. They are stored in the predefined and fixed *logical properties set P*. An example of such a predefined set P for the UML activity diagrams is shown in Fig. 2. Most elements of the P set, i.e. two temporal logic operators, classical logic operators, etc. are not in doubt. a_1, a_2 and a_3 are atomic formulas and constitute a kind of formal arguments for a pattern. The slash allows to place more than one formula in a single line. $c(a)$ means that the logical condition associated with the activity a has been evaluated and is satisfied. The pattern $SeqSeq$ means the concatenation of sequences as a sequence of three arguments. Variables in and out provide information about activities for a pattern that are the first and the last to be executed, respectively. They enable representing pattern to be considered as a whole. All formulas describe both safety and liveness properties for every pattern [15]. Summing up, the predefined set of pattern temporal properties consists of the following elements $\{Seq, SeqSeq, Flow, Switch, LoopWhile\}$ the meaning of which seems intuitive, i.e. sequence, sequence of a sequence, concurrency, choice and iteration.

Generating a logical specification is not a simple summation of formula collections resulting from a logical expression. The generation algorithm Γ_1 sketch for obtaining a set of temporal formulas is given below.

1) At the beginning, the logical specification is empty, i.e. $L := \emptyset$;
2) Patterns are processed from the most nested pattern to be located more towards the outside and from left to right;
3) If the currently analysed pattern consists only of atomic formulas, the logical specification is extended, by summing sets, by formulas linked to the type of pattern analysed, i.e. $L := L \cup pat()$;
4) If any argument is a pattern itself, then the logical disjunction of all elements that belong to in and out sets, is substituted in a place of the pattern;

The algorithm is a modification of the similar one presented in [13]. All patterns of the logical expression are processed one by one and the algorithm always halts. All parentheses are paired. The example of the algorithm is provided in the section V-B.

A logical specification L_1 consists of all formulas obtained from a logical expression using the algorithm Γ_1, i.e.

$$L_1(W_L) = \{f_i : i > 0 \wedge f_i \in \Gamma_1(W_L, P)\} \tag{2}$$

where f_i is a PLTL formula. The sketch of the generation algorithm is presented below. The generation process has two inputs. The first one is a logical expression and the second one is a predefined set of logical properties. The output is a set of logical formulas.

B. A rule-based approach

However, the rule-based approach for generating logical specifications is an interesting alternative for the previous one, i.e. presented in the section III-A. The approach seems suitable for the UML state diagram when considering set of states (nodes) and transitions (edges). The discussion is limited to some basic situations which are defined in terms of temporal logic formulas. Considering all transitions one by one, a logical specification understood as a set of temporal logic formulas is obtained using the following rules which constitute the generation algorithm Γ_2:

- R.1 (Sequence) – the state b is enabled when the state a is reached and an event e occurred, i.e.: $\{(a \wedge e) \Rightarrow \Diamond b, \Box\neg(a \wedge b)\}$;
- R.2 (Split) – when the state a is reached and an event e occurred then single thread of control is splited into two threads of control to enable parallel reaching the state b and the state c, i.e.: $\{(a \wedge e) \Rightarrow \Diamond b \wedge \Diamond c, \Box\neg(a \wedge (b \vee c))\}$;
- R.3 (Synchronization) – when two states a and b are reached in parallel and two events $e1$ and $e2$ occurred, respectively, then threads of control are transformed and synchronised into a single thread of control to enable reaching the state c, i.e.: $\{(a \wedge e1) \wedge (b \wedge e2) \Rightarrow \Diamond c, \Box\neg((a \vee b) \wedge c)\}$.

A logical specification L_2 consists of all formulas obtained using the algorithm Γ_2, i.e.

$$L_2 = \{f_i : i > 0 \wedge f_i \in \Gamma_2\} \tag{3}$$

where f_i is a PLTL formula. The input for the generation algorithm is a state diagram. The output is a set of logical formulas.

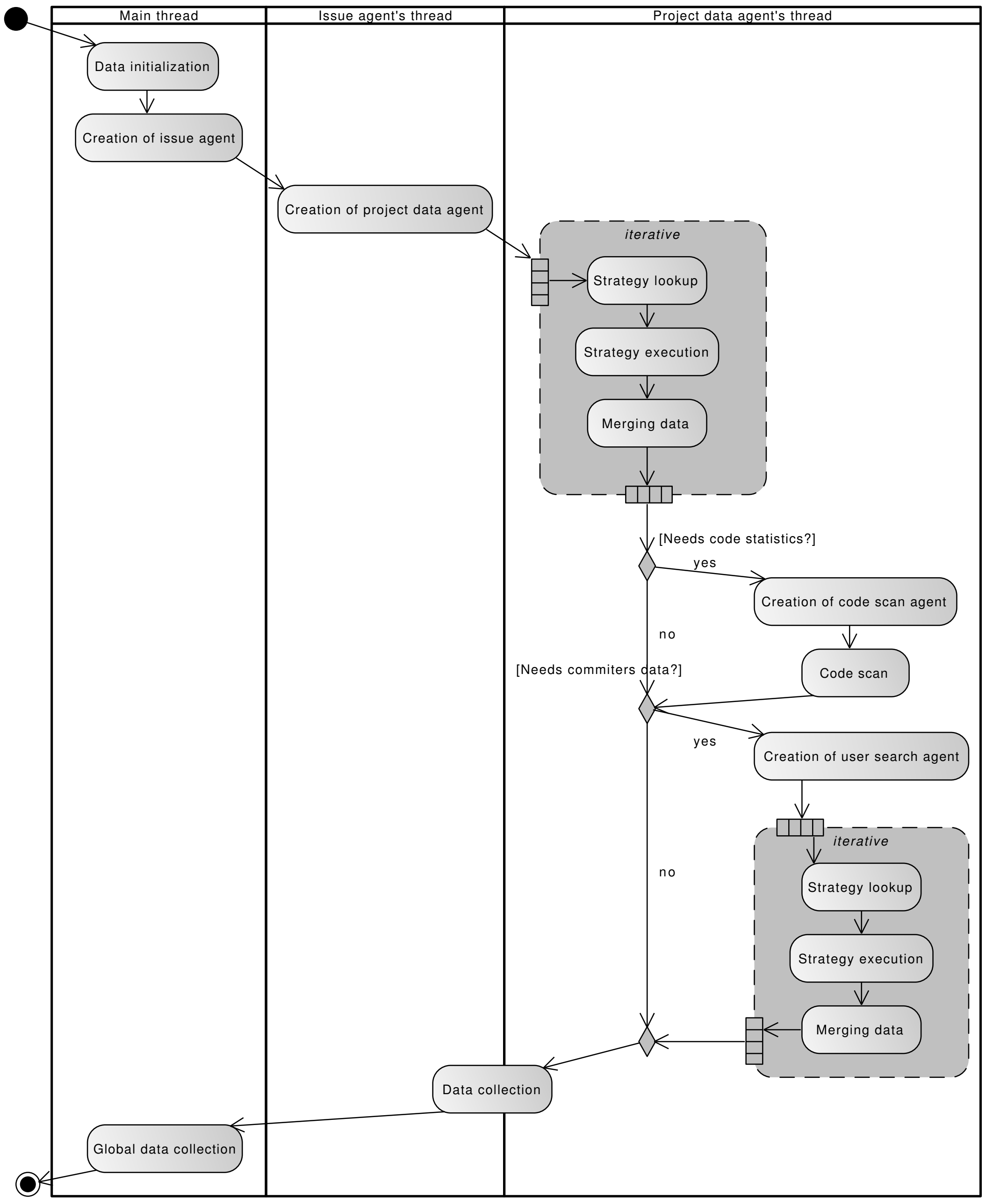

Fig. 3. Activity diagram of the search scenario presented in the section V-A.

IV. Agent-based framework for data integration

The vast amount of information available in the global network calls for complex systems able to perform various analyses with respect to data coming from various, often heterogeneous sources. The described framework provides the data- and task-oriented workflow for collecting and integrating data from a wide range of diverse services [4], [5]. Fig. 4 presents the layered structure of the framework:

- Presentation – the usage of different views is possible. They communicate with the rest of the system and allows the user to interact and control the act of the data integration.
- Middleware – the logical processing of the data is performed here. This part of the system uses the agent paradigm to delegate different parts of the data integration and processing to other parts of the system (or the external software, frameworks, etc.) that are represented also as agents.
- Services and data – wrapping of the various external data sources (and data processing capabilities) takes place here. Interfaces of the systems are adapted and described

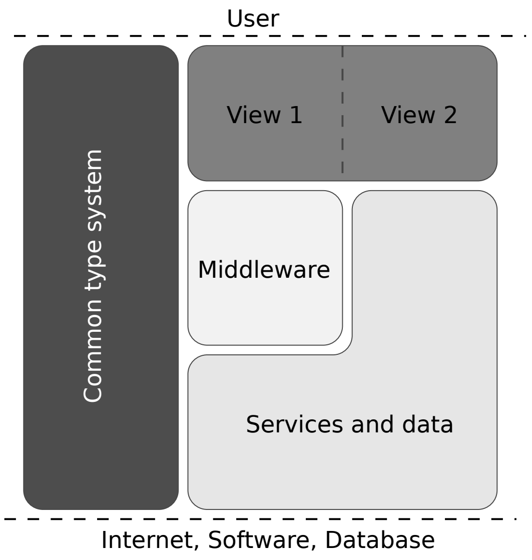

Fig. 4. Layered structure of the implementation

in terms of the common type system.
- Common type system – the whole framework uses the types described here to annotate the processed data, data sources and external systems interfaces or façades.

Queries created by the user are put into the agent system that performs two types of operations: the management of data (by inspecting queries and delegating them to other agents) and execution of demanded actions (including their selection, configuration and fault recovery). The system can divide processing into *issues*. An *issue* is a separate part of data processing, usually centred around an instance of a query object and all related (i.e. found) data.

The current implementation defines three possible functional roles for agents.

- **System agents** provide a bootstrapping and basic functionality. They create new issues, handle errors, monitor the system.
- **Issue Agents** care about a single issue (i.e. user query) and delegate initial tasks to specialised action agents. An Issue Agent retrieves a query from the pool, inspects it and then requests (with messages) a chosen data agent to resolve a query specified in the task.
- **Action (Data) Agents** implement the actual executive part of the search functionality. Upon receiving the task from an issue agent they obtain appropriate strategies and then executes them to answer the query.
  However, they are not constrained only to strategies. They can perform any action on data: merge, simplify, verify, etc.

Issue Agents are identified by a runtime-generated issue identifier that represents an issue hey are taking care of. The Action Agents are described during creation (implementation) with tasks they can perform (called “capabilities”) and data types they can operate on.

V. Data collection scenario and its formal analysis

We consider an act of gathering data about open-source projects as one of the possible use-cases. This kind of data can be easily mined from popular infrastructure-providing websites like GitHub, SourceForge, etc. Moreover, it is usually very well interlinked and it makes it possible to gather much broader information about e.g., people working on the project, organisations involved, etc.

A. Base Scenario

The scenario has three sample steps:

1) Gathering data about an open-source project from all available sources.
   The user inputs a query comprising of, for example, a name of the project or the link to its website into the system. The system performs a look-up of possible matches and returns a match of a list of possible matches. In the latter case, there may be a requirement to choose one (the best) match. It can be done manually or delegated to an agent that can rate each result and select the best one.
2) Scanning of the source code of the project.
   If possible, a user may expect the project's code to be scanned to obtain particular statistics. If expected, an agent responsible for project data may create another agent that can scan the code. Such an agent will perform the scan and merge the results back into the data handled by the project data agent.
3) Gathering information about commiters and authors.
   It may be further required to gather data about commiters and authors. The project data agent would create a user search agent that is able to collect users data from project website and other related sources.

On the system level our scenario is implemented as follows:

- The types like Project, Commiter, Code are introduced.
- The action agents that performs operations on types are implemented: Project Data Agent, Code Scan Agent, User Search Agent.
- Strategies for each external service can be created: Project Data Search and Code Scan for e.g., GitHub, SourceForge and User Data Search for, e.g., LinkedIn or Facebook.

Fig. 3 shows the activity diagram of an actual execution of the scenario. The user prepares a query that consists of the initial data to operate on (e.g. a name of the project). The task is placed into the system (*Data Initialization* action). Then, available issue agents are notified about it or a new one is created. The issue agent locates an implementation of a so-called action agent that can handle the specified task (Project Data Agent), instantiates it, and delegates the task execution to this agent.

Project Data Agent inspects the query and calls relevant strategies (to locate and gather project data). Then it may choose to instantiate the Code Scan Agent in order to scan the project's code. The same goes for commiters data – the

Project Data Agent can instantiate a specialised agent that will look up personal data.

Data collection is performed in two phases: first, the data from strategies is merged by a responsible agent (e.g., data about projects by the Project Data Agent); second, all data from an agent is inserted into the object provided by an agent higher in the hierarchy (e.g., Project Data Agent provides a Project object that has a field *commiters* that is filled by the User Search Agent). The query execution is finished by putting results to the pool presented to the user.

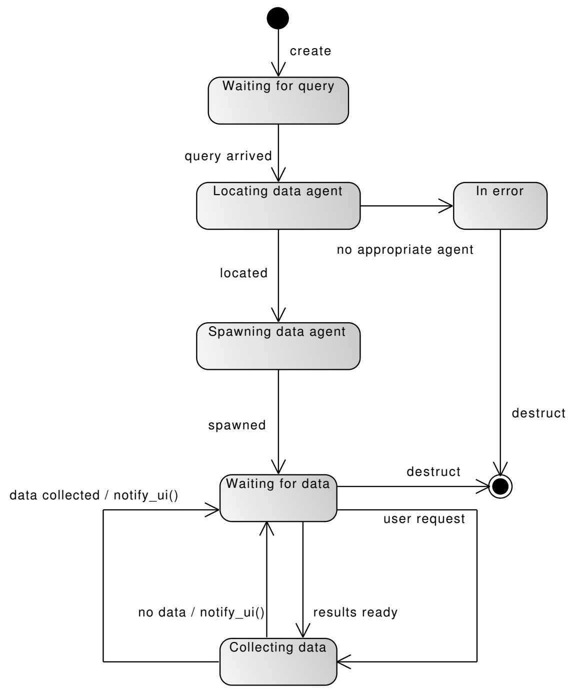

Fig. 5. State diagram of the Issue Agent in the search scenario presented in the section V-A.

Fig. 5 presents sample states of the Issue Agent that is responsible for initiating search for data and forwarding data to the user.

After creation and initialisation, an agent waits for a query from the user. When the query arrives, the agent tries to locate an implementation of a specialised data agent (in the scenario described earlier — Project Data Agent). If such an agent cannot be found it results in a fatal error as it is usually caused by a wrong configuration.

If the data agent is found it is instantiated by the Issue Agent and the query is forwarded to it. Then, the Issue Agent waits for either results from the data agent or request from the UI to get that data. It puts the data, if available, into the user pool and notifies the UI about it. If no data has been found, it generates an error for the UI.

B. Formal analysis and verification

Let us consider the activity diagram shown in Fig. 3. Diagram activities represent propositions which are used for modelling behaviour. Firstly, letters of the Latin alphabet are substituted in a place of propositions. Replacing atomic activities by Latin letters is a technical matter and is suitable only for the work because of its limited size. (In the real world, the original names of activities are used.) The following substitutions are made: a – DataInitialisation, b – CreationIssueAgent, c – CreationProjectDataAgent, d – ProjectDataLookup, e – NeedsCodeStatistics, f – CreationCodeScanAgent, g – CodeScan, h – NeedsCommitersData, i – CreationUserSearchAgent, j – UserSearchLookup, k – DataCollection, and l – GlobalDataCollection. (To simplify considerations, a single proposition instead of a loop is used.) The logical expression W_L for the activity diagram is

$$\begin{aligned}&SeqSeq(SeqSeq(a,b,c),SeqSeq(d,Switch(e,\\&Seq(f,g),N1),Switch(h,Seq(i,j),N2)),Seq(k,l))\end{aligned} \tag{4}$$

Activity diagrams constitute one of two inputs for the deduction system shown in Fig. 1. Two activities $N1$ and $N2$ are introduced since the diagram in Fig. 3 contains two switches without activity (null activity). A logical specification L_1 for the logical expression W_L is built using the algorithm Γ_1 presented in the section III-A. The logical specification is

$$\begin{aligned}L_1 = \{&f \Rightarrow \Diamond g, \Box\neg(f \wedge g), i \Rightarrow \Diamond j, \Box\neg(i \wedge j),\\&e \wedge c(e) \Rightarrow \Diamond(f \vee g), e \wedge \neg c(e) \Rightarrow \Diamond N1,\\&\Box\neg((e \wedge (f \vee g)) \vee (e \wedge N1) \vee ((f \vee g) \wedge N1)),\\&h \wedge c(h) \Rightarrow \Diamond(i \vee j), h \wedge \neg c(h) \Rightarrow \Diamond N2,\\&\Box\neg((h \wedge (i \vee j)) \vee (h \wedge N2) \vee ((i \vee j) \wedge N2)),\\&d \Rightarrow \Diamond e, e \Rightarrow \Diamond(j \vee N2),\\&\Box\neg((d \wedge e) \vee (e \wedge (j \vee N2)) \vee (d \wedge (j \vee N2))),\\&a \Rightarrow \Diamond b, b \Rightarrow \Diamond c, \Box\neg((a \wedge b) \vee (b \wedge c) \vee (a \wedge c)),\\&k \Rightarrow \Diamond l, \Box\neg(k \wedge l), (a \vee c) \Rightarrow \Diamond(d \vee i \vee j),\\&(d \vee i \vee j) \Rightarrow \Diamond(k \vee l),\\&\Box\neg(((a \vee c) \wedge (d \vee i \vee j)) \vee ((d \vee i \vee j) \wedge (k \vee l)) \vee\\&((a \vee c) \wedge (k \vee l)))\}\end{aligned} \tag{5}$$

Formula 5 represents the output of the generator component in Fig. 1.

Let us consider the state diagram shown in Fig. 5. Firstly, letters of the Latin alphabet are substituted in a place of states. The following substitutions are made: m – WaitingQuery, n – LocatingDataAgent, o – InError, p – SpawningDataAgent, q – WatingData, r – CollectingData, and s – Stop. Next, the following substitutions for events are made: $e'a$ – QueryArrived, $e'b$ – Located, $e'c$ – NoAgent, $e'd$ – Spawned, $e'e$ – Destruct, $e'f$ – UserRequest, $e'g$ – DataCollected, $e'h$ – NoData, and $e'i$ – ResultsReady. State diagrams constitute one of two inputs for the deduction system shown in Fig. 1. A logical specification L_2 is built using the algorithm Γ_2 presented in the section III-B. The logical specification is

$$\begin{aligned}L_2 = \{&(m \wedge e'a) \Rightarrow \Diamond n, \Box\neg(m \wedge n), (n \wedge e'b) \Rightarrow \Diamond p,\\&\Box\neg(n \wedge p), (n \wedge e'c) \Rightarrow \Diamond o, \Box\neg(n \wedge o),\\&(o \wedge e'e) \Rightarrow \Diamond s, \Box\neg(o \wedge s), (p \wedge e'd) \Rightarrow \Diamond q,\end{aligned}$$

$$\Box\neg(p \wedge q), (q \wedge e'e) \Rightarrow \Diamond s, \Box\neg(q \wedge s),$$
$$(q \wedge e'i) \Rightarrow \Diamond r, \Box\neg(q \wedge r), (q \wedge e'f) \Rightarrow \Diamond r,$$
$$(r \wedge e'h) \Rightarrow \Diamond q, (r \wedge e'g) \Rightarrow \Diamond q\} \quad (6)$$

Formula 6 represents the output of the generator component in Fig. 1.

Logical specifications which are built in the above way, i.e. using the proposed algorithms, can be verified formally. Formal *verification* is the act of proving correctness properties of a system. When the semantic tableaux method is used, then the whole reasoning process can be summarised as a process of the verification whether an entailment $F_1, ...F_n \models G$ it suffice to prove that $\{F_1, ..., F_n, \neg G\}$ is unsatisfiable. The *liveness* property of a system means that the computational process achieves its goals, i.e. something good eventually happens. The *safety* property of a system means that the computational process avoids undesirable situations, i.e. something bad never happens. The liveness property for the above model can be

$$b \Rightarrow \Diamond l \quad (7)$$

which means that **if creation of issue agent is satisfied then sometime global data collection is reached**, formally $CreationIssueAgent \Rightarrow \Diamond GlobalDataCollection$. When considering property expressed by formula 7 then the whole formula to be analysed using semantic tableaux, providing a combined input for the prover component in Fig. 1, is

$$C(L_1) \wedge C(L_2) \Rightarrow (b \Rightarrow \Diamond l) \quad (8)$$

where $C(L_x)$ means logical conjunctions of all formulas which belong to L_x, c.f. formula 5 and 6. Presentation of a full inference tree exceeds the size of the work. All branches of the semantic tree are closed, i.e. formula 7, is satisfied in the considered model. The method is easy to scale-up, i.e. extending and summing up logical specifications for other activity diagrams and state diagrams. Then, it is possible to examine logical relationships (liveness, safety) for different activities and states coming from different activity and state diagrams.

VI. Conclusions

The discussed agent-based data integration framework is based on AgE platform [16], which allows to introduce a clear separation of concerns and makes it possible to rapidly build topic-related versions of the system with little configuration. The framework is completely independent of the data it processes and all scenario-related components are provided by the user during the configuration. That is why the proposed verification approach seems to be particularly important to ensure proper configuration of the system.

Future work may include the implementation of the logical specification generation module and the temporal logic prover. Important results might be a CASE software for both the workflow modelling and the deduction-based formal verification. Considering graph transformations [17] is encouraging for models involving distributed representation of knowledge and their efficient implementation. The example of such an implementation is discussed in [18].

Acknowledgment

This work was supported by the AGH UST internal grant no. 11.11.120.859.

References

[1] N. R. Jennings, K. Sycara, and M. Wooldridge, "A roadmap of agent research and development," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, pp. 7–38, 1998.

[2] E. Clarke, J. Wing, and et al., "Formal methods: State of the art and future directions," *ACM Computing Surveys*, vol. 28 (4), pp. 626–643, 1996.

[3] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[4] E. Nawarecki, G. Dobrowolski, A. Byrski, and M. Kisiel-Dorohinicki, "Agent-based integration of data acquired from heterogeneous sources," in *Proc. of 5th Int. Conf. on Complex, Intelligent and Software Intensive Systems - CISIS 2011*, 2011.

[5] R. Klimek, Ł. Faber, and M. Kisiel-Dorohinicki, "Deduction-based modelling and verification of agent-based systems for data integration," in *Proceedings of 3rd International Conference on Man-Machine Interactions (ICMMI 2013), 22–25 October 2013, The Beskids, Poland [paper accepted]*, ser. Advances in Intelligent Systems and Computing, T. Czachórski, A. Gruca, and S. Kozielski, Eds. Springer Verlag, 2013.

[6] R. Klimek, "From extraction of logical specifications to deduction-based formal verification of requirements models," in *Proceedings of 11th International Conference on Software Engineering and Formal Methods (SEFM 2013), 25–27 September 2013, Madrid, Spain*, ser. Lecture Notes in Computer Science, R. Hierons, M. Merayo, and M. Bravetti, Eds., vol. 8137. Springer Verlag, 2013, pp. 61–75.

[7] F. Wolter and M. Wooldridge, "Temporal and dynamic logic," *Journal of Indian Council of Philosophical Research*, vol. XXVII(1), pp. 249–276, 2011.

[8] E. Emerson, *Handbook of Theoretical Computer Science*. Elsevier, MIT Press, 1990, vol. B, ch. Temporal and Modal Logic, pp. 995–1072.

[9] B. F. Chellas, *Modal Logic*. Cambridge University Press, 1980.

[10] J. van Benthem, *Handbook of Logic in Artificial Intelligence and Logic Programming*, ser. 4. Clarendon Press, 1993–95, ch. Temporal Logic, pp. 241–350.

[11] M. d'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, *Handbook of Tableau Methods*. Kluwer Academic Publishers, 1999.

[12] R. Klimek, "Temporal preference models and their deduction-based analysis for pervasive applications," in *Proceedings of 3rd International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2013), 19–21 February, 2013, Barcelona, Spain*, C. Benavente-Peces and J. Filipe, Eds. SciTePress, 2013, pp. 131–134.

[13] ——, *Advanced Methods and Technologies for Agent and Multi-Agent Systems*, ser. Frontiers of Artificial Intelligence and Applications. IOS Press, 2013, vol. 252, ch. A Deduction-based System for Formal Verification of Agent-ready Web Services, pp. 203–212. [Online]. Available: http://ebooks.iospress.nl/publication/32843

[14] T. Pender, *UML Bible*. John Wiley & Sons, 2003.

[15] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21 (4), pp. 181–185, 1985.

[16] L. Faber, K. Piętak, A. Byrski, and M. Kisiel-Dorohinicki, "Agent-based simulation in age framework," in *Advances in Intelligent Modelling and Simulation*, ser. Studies in Computational Intelligence, A. Byrski, Z. Oplatková, M. Carvalho, and M. Kisiel-Dorohinicki, Eds. Springer Berlin Heidelberg, 2012, vol. 416, pp. 55–83. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28888-3_3

[17] L. Kotulski, "Supporting software agents by the graph transformation systems," in *International Conference on Computational Science*, ser. Lecture Notes in Computer Science, V. N. Alexandrov and et al., Eds., vol. 3993. Springer, 2006, pp. 887–890.

[18] I. Wojnicki, L. Kotulski, and S. Ernst, "On scalable, event-oriented control for lighting systems," *Frontiers in Artificial Intelligence and Applications: Advanced Methods and Technologies for Agent and Multi-Agent Systems*, vol. 252, pp. 40–49, 2013.