Situational Awareness for Industrial Operations

Peter Baumgartner and Patrik Haslum*
CSIRO/Data61 and Australian National University

Abstract

The smooth operation of industrial or business enterprises rests on constantly monitoring, evaluating and projecting their current state into the near future. Such *situational awareness* problems are not well supported by today's software solutions, which often lack higher-level analytic capabilities. To address these issues we propose a modular and re-usable system architecture for monitoring systems in terms of their state evolution. As a main novelty, states are represented explicitly and are amenable to external analysis. Moreover, different state trajectories can be derived and analyzed simultaneously, for dealing with incomplete or noisy input data. In the paper we describe the system architecture and our implementation of a core component, the state inference engine, through a shallow embedding in Scala. The implementation of our modeling language as an embedded domain-specific language grants the modeler expressive power and flexibility, yet allows us to abstract a significant part of the complexity of the model's execution into the common inference engine core.

1 Introduction

The smooth operation of industrial or business enterprises like supply chains, assembly lines or warehouses, critically depends on maintaining situational awareness. In our context, *situational awareness* is the problem of gathering changes in the operation's environment, aggregating them for deriving the current state of the operation at a semantically high level, projecting the current state into the near future, alerting the user of potential problems, and proposing corrective action if required.

Situational awareness in this sense currently lacks software tool support. While database systems and ERPs can help human operators gain situational awareness, they do not offer a complete solution with integrated analytic capabilities. In this paper we address this issue by proposing an architecture for a novel situational awareness software platform.

This architecture has its origin in a situational awareness and scheduling system for the factory floor of an industrial client, which we generalize to be applicable in a variety of domains, ranging from local operations, such as factory floor assembly lines or warehouse management, to large scale and distributed operations, such as

*This research is supported by the science and industry endowment fund.

national or international supply chains or the Array of Things [AoT]. We also describe a concrete realization of one core component, the state inference engine.

The state inference engine maintains a current state which is updated whenever an external event comes in. In the current implementation, state update is described in terms of rules that are triggered by such events. It offers a modeling paradigm that assigns rule sets to (parallel) processes, one process for each entity of interest, and message channels connecting these processes. The execution of the processes' rules enables inference about aspects of the state that are not directly observable, including disjunctive reasoning about alternative states and execution histories.

While the model is domain-dependent, the interpreter is universal. It is, in essence, a forward-chaining rule engine and is realized via a shallow embedding in Scala. The main advantage of this design is that it gives access to the full power of the Scala language to represent process states and express conditions, functions and transformations on them. It provides expressive power and flexibility, at little implementation cost. It is this modeling language and inference engine implementation that we put forward as the main contribution of this paper. Due to its modular design, our architecture allows for transparently substituting another method of state inference, e.g., conflict-directed diagnosis [GHT12], with its associated modeling formalism.

We emphasize that we do not intend to replace existing systems, e.g., ERPs or tailored scheduling systems. Instead we want to capitalize on their functionality by integrating and augmenting them with inference capabilities for the purpose of gaining a more holistic view. An important feature of our system is the explicit and externally analyzable state representation, enabling it to simultaneously represent and explore different state trajectories, for example to assess their plausibilities and what-if reasoning. This ability is essential to deal with noisy or unreliable data sources.

2 Related Work

Supply chains have become complex networks with automated transitions to manage the manufacturing and flow of goods. Techniques like service oriented architectures (SOAs) and business process modeling (BPM) [van13] are instrumental for automating and optimizing supply chains with respect to life cycle management, procurement, logistics, and order management. BPM comprises methods for making business operations explicit, such as BPMN [OMG13], which is intended mainly for manual use. Such explicit models are typically not designed from the ground up for situational awareness, but they may well inform models for situational awareness. Software designs like SOAs or model-driven architectures do not lead to situational awareness by themselves, but may integrate situational awareness capabilities.

The basic principle of Industry 4.0 is to connect machines, work pieces and business management systems in intelligent networks for controlling each other autonomously. Examples are machines that can predict failures and trigger maintenance processes autonomously, or self-organized logistics which react to unexpected changes in production [Ind]. Companies like Oracle and SAP have proposed to go further and leverage the new technologies for *situational awareness*. The underlying observation is that automation is not enough, the overall system should be conscious and knowledgeable of its surroundings [Obe17]. Indeed, some commercial "business intelligence" information systems [SAP17] offer event management functionality that allows users to

monitor and measure supply chain activities. However, existing systems lack the capability to perform deeper inference, for instance about unobserved events or completion of missing data.

Derigent and Thomas [DT17] expressed the need for situational awareness in the context of the Internet of Things (IoT). They propose a corresponding architecture and identify key functionalities. Similar proposals were made by Lee, Ardakani, Yang and Bagheri [LAYB15], and by Singh and Tripathi [STJ14]. They all remain somewhat inconcrete regarding the realization of their proposals. Ghimire, Luis-Ferreira, Nodehi and Jardim-Goncalves [GLFNJG17] go further and mention a situational awareness module that makes use of formal knowledge representation and semantic web technologies such as OWL (Ontology Web Language). From their description, however, it seems that the module is currently under implementation and no instantiation of their architecture is yet available.

One of the key uses of formalized business models is *monitoring* their execution for conformance to the model. Cook and Wolf [CW99] developed techniques for uncovering and measuring the discrepancies between models and executions. They utilize rather high-level metrics for that. In contrast, and closer to our work, Chesani, Mello, Montali, Riguzzi, Sebastianis and Storari [CMM+08] propose a framework for performing conformance checking of process execution traces w.r.t. expressive reactive business rules. Rules are mapped to Logic Programming, using Prolog to classify execution traces as compliant/non-compliant. Our approach is rule-based as well but additionally offers processes and channels as a modeling paradigm. DeGiacomo, Maggi, Marella and Sardina [DMMS16] also address conformance checking, by modelling process rules in the declarative Planning Domain Definition Language (PDDL) and applying an off-the-shelf automated planner. They also attempt to identify which are the missing or superfluous activities in non-conformant traces, which is a special case of the general problem of diagnosis of discrete dynamical systems [CT94, McI94].

Formal modeling languages, such as Petri nets or the Promela language used by the Spin model checker [Hol97], are used for modeling business processes or workflows as well as concurrent processes in (embedded) computer systems. However, these languages do not support modeling processes whose internal state or messages are made up of complex data types.

In the runtime verification area, Kauffman, Havelund and Joshi [KHJ16] propose a more specialized Scala DSL for monitoring event streams over Allen's temporal interval logic [All83]. For a different application, Havelund and Joshi [HJ17] propose a Scala DSL based on hierarchical state machines. These approaches overlap with our approach in terms of implementation (Scala DSL) but differ conceptually. In particular, our approach supports explicit candidate model computation via disjunctive rules and backtracking.

Outline of the paper. The rest of this paper is structured as follows: in Section 3 we present an overview of our system architecture. This is needed to provide context for our process modeling language and how it is executed. The former is the subject of Section 4 and the latter is explained in Section 5 on the inference engine. We conclude in Section 6 with a brief summary and future work.

3 System Architecture

The system architecture is depicted in Figure 1. Its components may run in parallel, and may be distributed across sites. We use the term "message" for the information flowing between the components.

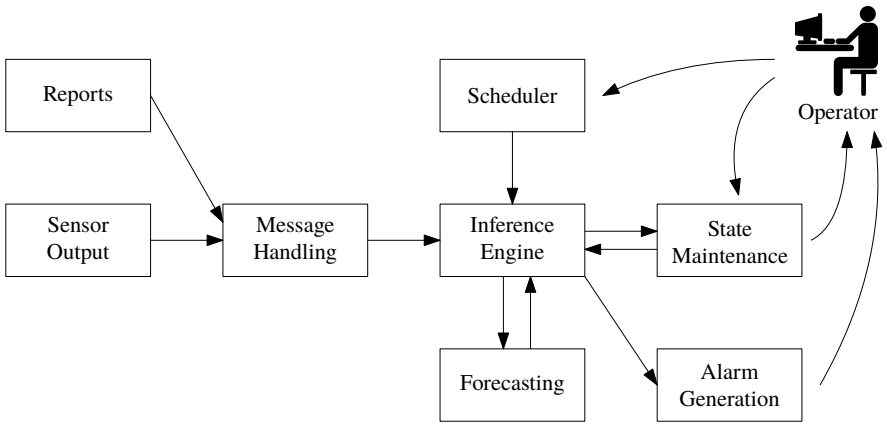

Figure 1: System architecture.

Reports, sensor output and message handling The situational awareness system is driven by and reacts to external events. These may come from sensors, such as RFID and video (e.g., on a factory floor) or they may be reports filed by humans (e.g., orders and dockets in a sales business).

We do not assume that the environment provides complete information, e.g., sensors can fail and workers forget to report, or report incorrectly. However, we assume an abstraction mechanism, the "Message Handler" which preprocesses input messages into a uniform message format. It may also carry out other operations, such as adding a timestamp, sorting, or filtering.

Inference engine In addition to sensor and report input being unreliable, one cannot expect a complete model of the world under consideration. It is the task of the inference engine to use the information at a given time to derive a more complete description of the current overall state. This may involve inferring missing information, what-if reasoning and auto-correction.

The inference engine is general and needs to be equipped with a concrete model. In our current realization, the model is given by processes connected via message channels. Execution is stateful, with user-supplied rules governing state transitions. We describe this in detail in Section 5 below. For now, it suffices to say that the inference engine receives a windowed sequence of message, and, as a result, derives one or more plausible current states. Analyzing this state or states in context with the state history is the responsibility of the state maintenance module. Figure 2 gives an overview of these dynamic aspects of the inference engine and the state maintenance module.

State maintenance The state maintenance module manages a set of *execution paths*, or just *paths*. A path is a time-finite sequence of states obtained by successive runs of

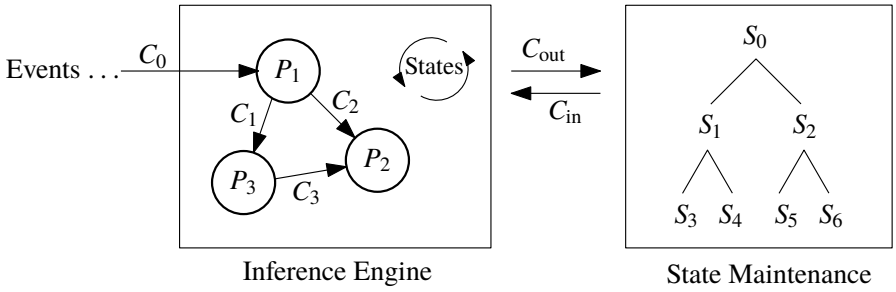

Figure 2: Inference engine and state maintenance. Events are coming in from the channel C_0. The inference engine has processes P_1, P_2, P_3 and internal channels C_1, C_2, C_3. The channels C_{in} and C_{out} connect the two components for exchanging states. The state maintenance module maintains possible execution paths in terms of states derived by the inference engine.

the inference engine (details in Section 5 below). The set of execution paths naturally forms an *execution tree*, cf. again Figure 2. The execution tree is meant to represent possible executions of the modeled system. The state maintenance constructs the execution tree based on new states coming in from the inference engine, and it informs the inference engine by telling it a next state to continue with. We emphasize that this next state does not have to be among the input states, it could be any state derived from the execution tree so far.

The full generality of execution trees may not be needed in all applications. Some immediate special cases and variations come to mind:

- *Markovian:* no histories; every branch is a singleton, the most recent state.
- *Deterministic:* the tree has only one branch.
- *Focused:* only some branches, e.g., the most plausible states, are kept.
- *Probabilistic:* assign each state a probability distribution over its children.

While state maintenance could in principle be done by the inference engine, there are arguments to keep it separate. Because the nodes along a path are temporally ordered, execution trees can be analyzed with temporal logic. One would express (un)desirable properties with logics such as linear temporal logic (LTL), computational tree logic (CTL) and apply verification methods to them. (See [BK08] for a textbook on temporal verification.) Particularly relevant are runtime verification methods capable of handling structured data like those developed by Havelund and Peled [HP18]. For instance, one could formulate a temporal LTL safety constraint for monitoring travel times between waypoints for fresh goods and raise an alarm if violated. Also, techniques for log analysis seem applicable. For instance, Brandt et. al. [BKR+18] propose an expressive ontology-based framework on top of metric temporal logic.

Scheduler The scheduler provides the timeline for future events so that a mission can be accomplished. In a supply chain domain it could be route planning, among others (we consider the term "Scheduler" loosely).

Forecasting The components described so far are backwards-looking in time. The forecasting module is concerned with projecting the current state into the near or medium future, and thus requires additional information, e.g., the current schedule

stored in a database. Even a simple approximate prediction can be useful in practice. In a supply chain domain, for instance, forecasting may combine current vehicle locations with the routes yet to be traveled.

Alarm generation Generating alarms or notifying the user of deviations of the expected state is a core functionality of situational awareness systems. In our current implementation, this functionality is provided by the inference engine, which is poised for doing this as it has at its disposal (a) the current state, (b) the current schedule, and (c) the forecast. Incoherencies between (a), (b) and (c) are thus detectable. Conceptually, however, alarm generation is a different activity from state inference, as it involves a judgment about whether the state merits human attention, and not only what the state is.

Operator The operator receives notifications from the system, in particular alarms, and interacts with the state maintenance system to achieve faithful state representation. The operator invokes scheduling when needed, e.g., in case of alarms.

4 The Process Modeling Language

The main rationale behind our modeling language is to provide a natural framework for mapping actors in the real world – physical or conceptual ones, like schedules – to corresponding modeling entities. Our modeling approach supports object-oriented design principles like abstraction, polymorphism, encapsulation and inheritance. The main entities are *processes*, which run (conceptually) in parallel and exchange information through *messages*. The message passing paradigm was inspired in part by the Spin model checker and its modeling language Promela [Hol97]. Spin is geared towards model checking, i.e., the problem of proving or disproving that all possible runs of the system satisfy some property. Unlike Spin, our system is tailored towards situational awareness, which analyzes *one* run (the "reality") and it supports object-oriented design principles.

Processes *Processes* are models of the entities of the system at hand. The entities could be physical entities, such as machines producing goods and workers scheduled for work, or abstract entities, such as schedules and rosters. Processes can be created and scheduled for execution dynamically, and they can be stopped and destroyed dynamically. Processes are stateful, where a process' *state* is given by a user-declarable set of local variables and their current values. It is the collection of the states of all processes that is sent to the the state maintenance component at certain times.

The computation inside a process is described by a set of *rules*. A rule is an if-then statement whose condition may involve reading a channel. If a message is available at the time and the rule's condition is met then the rule is *executable*. Otherwise the rule *fails*. The conclusion of a rule typically modifies local variables and/or sends data to other processes. The conclusion can be disjunctive, which allows for branching into alternative possible states that may be consistent with the current, incomplete state. The details of process execution are below in Section 5.

Messages and channels *Channels* are used for sending *messages* into and out of processes. Messages can be of any type as long as they can be serialized. We distinguish between *internal* and *external* channels. Internal channels connect processes, while

external channels connect processes with the outside world.[1] External channels can be *frozen*, which means that incoming data is deferred into a buffer until the channel is unfrozen again. Channels are lossless and can be used for *m*-to-*n* communication. Any number of processes can *subscribe* to a channel. This has the effect that incoming messages are duplicated to all subscribers, so that multiple processes can receive and deplete the channel individually without interfering with each other, cf. Figure 3.

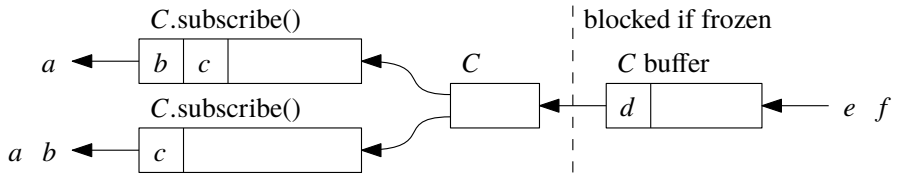

Figure 3: The channel subscription model. In this example, an external channel *C* has already received data items *a*, *b* and *c*. Then *C* was frozen and incoming data *d* was put into the buffer for now while *e* and *f* have not arrived yet. The data *a*, *b* and *c* already received is actually copied and forwarded to all subscribing channels. In the example, the upper subscribing channel has *a* being read from, and the lower subscribing channel has *a* and *b* being read from.

4.1 Illustrative Example

As an example, we consider a highly simplified food supply chain, which we model the supply as a set of interacting processes. A process may correspond to an actor in the supply chain (such as a distributor), or an actor may be modeled as several interacting processes (for example, a distributor may be decomposed into inventory, transportation and contracting processes). It is also possible for processes to model interfaces between actors (such as a shipment). Processes act according to their internal logic, which is codified in the model, and interact via message channels. The channel mechanism is necessary to model the synchronization of processes (e.g., between a delivery and an inventory process).

Processes There are *goods* (apples and oranges) of specific *origin* (Riverina and Batlow) which are transported between *warehouses* (Sydney, Goulburn, Canberra) by *trucks* (TruckA, TruckB and TruckC). "Truck" and "Warehouse" are process *classes* in our model, and each truck and each warehouse is an instance of its class.[2]

Messages and channels Occasionally, *waypoints* for the trucks are available (say by GPS) in terms of time and location. There are *dockets* for goods and their origins, and the goods are loaded on trucks at a warehouse. These events are sent as messages into an external input channel and then dispatched into internal channels typed for "waypoint" notifications and "loading" activities.

Process rules The process logic supports that waypoints are not noticed and that occasionally the origin of goods is not recorded. That is, not all events find their way into the system, or some remain partially specified . In general, the system should

[1] External channels must be equipped with deserialization for their message type.
[2] As an object-oriented language, Scala gives us the class/instance paradigm for free.

try its best to complete missing information or deal with it in another reasonable way. This is obviously a domain-dependent task. For the sake of illustration, we use the following rules for trucks:

1. If a current waypoint message specifies a location L for truck T then L is recorded as T's current location.
2. If a current loading message specifies that goods have been loaded on truck T at location L and the current location recorded with T is different to L then a waypoint message is broadcast, specifying that T is at L.
3. If a current loading message specifies that goods have been loaded on truck T with "unknown" origin then (a) that loading message with "unknown" is replaced by "Riverina" or (b) that loading message with "unknown" is replaced by "Batlow" through broadcasts.

Rule (1) is the normal way of recording locations of trucks. Rule (2) infers a missing waypoint message. Notice that Rule (2) does not simply set the current location of T to L but sends a message instead. This allows other processes listening for waypoint messages (the warehouse processes, in our example) to also be informed about the inferred waypoint. Rule (3) branches into alternatives for resolving the missing information by making it concrete. Each case will be investigated in a separate strand of computation and may trigger further consequences ("what-if" reasoning).

4.2 Shallow Embedding in Scala

Our approach to process modeling is implemented in Scala [Sca]. Scala is a modern high-level programming language that combines object-oriented and functional programming styles. Scala comes with a comprehensive library (e.g., for container data structures) and runs on the Java virtual machine, allowing Scala code to use existing Java libraries in a straightforward way.

Scala has functions as first-class objects and supports user-definable pre-, post- and infix syntax. With these features, Scala is suitable as a host language for embedding domain-specific languages (DSLs). (See, e.g., [HJ17] for a Scala DSL for runtime verification.) In our case, we embed a DSL for modeling the processes and channels from above. The embedding is shallow, i.e., the source constructs of the DSL are actual Scala code with DSL-specific classes and methods, which then needs to be compiled by the Scala compiler to be executable. Scala's (optional) call-by-name parameter passing style enables us to take statements as data and, hence, to explicitly invoke their execution, or not. This feature was instrumental for implementing rules as partial functions, where rule applicability reduces to partial function definedness and rule execution reduces to statement execution.

As a drawback, the shallow embedding approach makes it harder to analyze DSL expressions, e.g., for errors, and statements are executed as "black boxes". We found the advantages outweigh the drawbacks though. The DSL includes the full host language, meaning we can use Scala objects and classes without compromising efficiency. By contrast, a deep embedding would require us to write an interpreter for the full DSL. Instead of going into the details of the implementation we illustrate with some DSL source code.

Listings 1, 2 and 3 show concrete excerpts of our food supply chain example written in our DSL. A shallow embedding, it is comprised mostly of standard Scala. The DSL specific language constructs are underlined.

Listing 1: Channels and message types. **package** and include declarations are not shown. Here and below, the dots indicate "glue code" for serialization and deserialization.

```
1  // Waypoint: observed time and location of a specific truck
2  case class Waypoint(time: LocalDateTime, truck: String, location: String)
3  object WaypointChannel extends Channel[Waypoint]("Waypoint") { ... }
4
5  // Loading: time and location of goods from a specific origin loaded on a truck
6  case class Loading(time: LocalDateTime, truck: String, location: String,
7                     goods: String, origin: String)
8  object LoadingChannel extends Channel[Loading]("Loading") { ... }
9
10 // Input: sole external channel for receiving messages in Json
11 object Input extends Channel[JsObject]("Input", withInputPort = 5554, window = 1)
```

Listing 1 defines the main message types – Waypoint and Loading – and corresponding internal channels. The message types are ordinary Scala case classes. Input is an external channel whose messages are deserialized and dispatched into the other two channels. We use JSON as the format of external messages, but any other format can be used in its place. The declaration withInputPort=5554 specifies that the channel's messages are received over TCP. The window size says how many messages are taken from the input queue for the next round of processing.

Listing 2: The Truck processes, one for each truck.

```
1  class Truck(Id: String) extends Process("Truck") {
2    // Keeps track of the current location and load of this truck
3    var location = "unknown"
4    var load = Set.empty[(String, String)] // Items on this truck, as (goods, origin)
5
6    stateVar("location", ..., ...) // The Process state is comprised of location and load
7    stateVar("load", ..., ...)
8
9    val waypointChannel = WaypointChannel.subscribe() // Channel subscriptions
10   val loadingChannel = LoadingChannel.subscribe()
11
12   rules (
13     waypointChannel --> {
14       case Waypoint(_, Id, loc) if location != loc =>
15         location = loc // Update current location
16       case _ => () // All other cases ignored
17     },
18     loadingChannel --> {
19       case p @ Loading(time, Id, loc, _, _) if location != loc =>
20         // Infer waypoint from this Loading message and inform all processes
21         WaypointChannel <-- Waypoint(time, Id, loc)
22         LoadingChannel <-- p // Send Loading message again in order not to loose it
23       case Loading(time, Id, loc, goods, origin) if origin != "unknown" =>
24         // Fully specified loading
25         load += ((goods, origin)) // Add to current load
26       case Loading(time, Id, loc, goods, origin) if origin == "unknown" =>
```

```
27          // Partially specified loading
28          // Branch out into two cases, replacing unknown origin by concrete places
29          or ( LoadingChannel <-- Loading(time, Id, location, goods, "Riverina"),
30               LoadingChannel <-- Loading(time, Id, location, goods, "Batlow") )
31        case _ => ()
32      }
33    )
34  }
```

Listing 2 defines the Truck process class. The main program (not shown here) schedules instances by statements like Scheduler.schedule(**new** Truck("TruckA")). A truck's state is given by its location and current load. Lines 3 and 4 are the local variables, lines 6 and 7 declare them as the process' externally visible state. Lines 9 and 10 subscribe to the two channels of interest for the process. The rules-statement in line 12 defines two rules.

The first rule reads the WaypointChannel via the `-->` method. A **case** statement defines a partial function by pattern matching. The first case sets the current location to the location given by the Waypoint message. The second case is a catch-all to make sure that the channel will not be blocked if the first case does not apply.

The second rule deals with Loading messages. The first case infers a (possibly missing) Waypoint message from the Loading message and sends it to the Waypoint channel. It will be picked up later by the first rule and processed as described above. The second case applies to complete Loading messages, and updates the load variable. The third case applies when the origin of the goods is "unknown". By its disjunctive conclusion, the or-statement, state generation branches out by replacing "unknown" with concrete alternatives.

Listing 3: The Warehouse processes, one for each warehouse.

```
1  class Warehouse(Location: String) extends Process("Warehouse") {
2    // Keeps track of the set of trucks currently at this warehouse
3    var trucks = Set.empty[String]
4    stateVar("trucks", ..., ...)
5
6    val waypointChannel = WaypointChannel.subscribe()
7    rules (
8      waypointChannel --> {
9        case Waypoint(time, truck, Location) if
10           (! (trucks contains truck)) => trucks += truck
11       case Waypoint(time, truck, loc) if loc != Location &&
12           (trucks contains truck) => trucks -= truck
13       case _ => ()
14     }
15   )
16 }
```

Finally, Listing 3 shows a second process class that subscribes to the Waypoint channel. A Warehouse process instance reads Waypoint messages to track which trucks are currently at the warehouse. Note that this process will also see Waypoint messages inferred by a Truck process.

5 Inference Engine

The inference engine executes the models written in our language. This is done in rounds of scheduling and running its processes. Figure 4 shows the implemented algorithm as pseudo-code.

At the beginning of each round, all external channels are frozen, so that incoming messages are temporarily buffered, cf. Figure 3. The *scheduler* then selects in a fair way the next process for execution among the scheduled processes. This process is executed by trying its rules, once, in the given order. A failed rule application has no effect, i.e., it never modifies a state or consumes channel messages. The conclusion of the first executable rule is executed. If the conclusion is a disjunction, the first alternative is executed and the second alternative is put away together with the current state, and later restored for execution. That is, disjunctions require maintaining a tree of execution sequences over saved states and channel data. The execution of a conclusion (case) may also explicitly fail.[3] In this case process and channel states are restored to what they were before.

In each round, this selection of processes is repeated until all channels are depleted or a user-defined cutoff number of rule executions has been reached. This results in one or more derived states. These are sent, via dedicated external channel, to the *state maintenance* module, whose task is to derive from them some state that is sent back to the inference engine as the new current state for the next round. Finally, the external channels are unfrozen so that messages arrived in the meantime become available, and the next round starts.

6 Conclusions

We introduced a novel architecture for situational awareness for industrial operations. As our main contribution in this paper, we presented a design and implementation of one core component, the inference engine and its associated process modeling language, via a domain-specific embedding into Scala. We illustrated our approach with a small example from the food supply chain domain. Our implemented system runs this example as described in the main part of the paper, but we did not include a log here for space reasons. We have also been gathering experience with our system on more elaborate food supply chain, factory floor, and data cleaning applications. In each of these, we have found our modeling approach of processes, rules and channels confirmed to be viable in practice. Notwithstanding this promising experience, we need to further mature our system as a prerequisite for wider impact. We envisage a number of things: enriching the modeling language by an ontological component, e.g., a description logic, for added declarative domain modeling and reasoning; employing a declarative, temporal-logic based system for state maintenance as indicated in Section 3; model-checking the process models (this will be possibly only for controlled subsets of Scala); and probabilistic reasoning based on distributions for conclusions of disjunctive rules.

[3] Similar to Prolog's `fail` statement.

References

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, November 1983.

[AoT] The Array of Things. `https://arrayofthings.github.io/`.

[BK08] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[BKR+18] Sebastian Brandt, Elem Guezel Kalaycı, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Querying Log Data with Metric Temporal Logic. *Journal of Artificial Intelligence Research*, 62(5):829–877, August 2018.

[CMM+08] Federico Chesani, Paola Mello, Marco Montali, Fabrizio Riguzzi, Maurizio Sebastianis, and Sergio Storari. Checking compliance of execution traces to business rules. In *International Conference on Business Process Management*, Springer, 2008.

[CT94] M.O. Cordier and S. Thiébaux. Event-based diagnosis for evolutive systems. In *Proc. 5th Int'l Workshop on Principles of Diagnosis*, 1994.

[CW99] J. E. Cook and A. L. Wolf. Software process validation: Quantitatively measuring the correspondence of a process to a model. *ACM Transactions on Software Engineering Methodology*, 8(2), 1999.

[DMMS16] Giuseppe De Giacomo, Fabrizio Maria Maggi, Andrea Marrella, and Sebastian Sardina. Computing trace alignment against declarative process models through planning. In *Proc. ICAPS*, 2016.

[DT17] William Derigent and André Thomas. Situation awareness in product lifecycle information systems. In *Service Orientation in Holonic and Multi-Agent Manufacturing - Proceedings of SOHOMA 2017*, Springer, 2017.

[GHT12] Alban Grastien, Patrik Haslum, and Sylvie Thiébaux. Conflict-based diagnosis of discrete event systems: Theory and practice. In *Proc. KR'12*, 2012.

[GLFNJG17] Sudeep Ghimire, Fernando Luis-Ferreira, Tahereh Nodehi, and Ricardo Jardim-Goncalves. Iot based situational awareness framework for real-time project management. *International Journal of Computer Integrated Manufacturing*, 30(1), 2017.

[HJ17] Klaus Havelund and Rajeev Joshi. Modeling rover communication using hierarchical state machines with scala. In *Computer Safety, Reliability, and Security - SAFECOMP 2017 Workshops*, LNCS 10489, Springer, 2017.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[HP18] Klaus Havelund and Doron Peled. Efficient runtime verification of first-order temporal properties. In *Model Checking Software - 25th International Symposium, SPIN 2018*, LNCS 10869, Springer, 2018.

[Ind] Industry 4.0. `https://en.wikipedia.org/wiki/Industry_4.0`.

[KHJ16] Sean Kauffman, Klaus Havelund, and Rajeev Joshi. nfer - A notation and system for inferring event stream abstractions. In *Runtime Verification - 16th International Conference, RV 2016*, LNCS 10012, Springer, 2016.

[LAYB15] Jay Lee, Hossein Davari Ardakani, Shanhu Yang, and Behrad Bagheri. Industrial big data analytics and cyber-physical systems for future maintenance & service innovation. *Procedia CIRP*, 38:3 – 7, 2015.

[McI94] S. McIlraith. Toward a theory of diagnosis, testing and repair. In *Proc. 5th Int'l Workshop on Principles of Diagnosis*, 1994.

[Obe17] Supreet Oberoi. Situational Awareness for the Factory Floor, 2017. `https://blogs.oracle.com/iot/situational-awareness-for-the-factory-floor`.

[OMG13] OMG. *BPMN 2.0: Specification*, 2013. `www.omg.org/spec/BPMN/`.

[SAP17] *SAP Event Management 9.2*, 1.8 edition, July 2017.

[Sca] The Scala Programming Language. `https://www.scala-lang.org`.

[STJ14] Dhananjay Singh, Gaurav Tripathi, and Antonio J. Jara. A survey of internet-of-things: Future vision, architecture, challenges and services. In *2014 IEEE World Forum on Internet of Things, WF-IoT 2014*. IEEE, 2014.

[van13] Wil M.P. van der Aalst. Business process management: A comprehensive survey. *ISRN Software Engineering*, 2013.

Figure 4: Pseudo-code of the inference engine. Some details are left out, e.g., forced termination of the inner while-loop and statements that dynamically create or destroy processes. The definedness test on line 15 and code execution on line 19 is possible thanks to Scala's higher-order features.

```
1  procedure inferenceEngine
2
3  function save() = "a snapshot of all current states and channels"
4  procedure restore(s) "restore states and channels from a saved() snapshot s"
5
6  var btp := {} // set of pairs ⟨snapshot,statement⟩ "backtrack points"
7  while true do
8     freeze all external channels
9     var res := {} // set of snapshots computed in current round
10    while some channel is non-empty do
11       let proc = chose process among scheduled processes
12       // Find first executable rule and execute it
13       for (p ← proc.rules) do
14          let (ch ⟶ pf) = p // channel ch and partial function pf
15          if (ch.nonEmpty) ∧ pf(ch.first) is defined then
16             let s = save() // needed if pf(ch.first) fails
17             let d = ch.dequeue() // destructively remove first element
18             try
19                execute pf(d)
20                // if pf(d) executes or(stmt1,stmt2) then
21                //       btp := btp ∪ { ⟨save(),stmt2⟩}
22                //       execute stmt1
23                // endif
24                break // for-loop
25             catch case Fail =>restore(s)
26          end // if
27       end // for
28    end // while
29    res := res ∪ { save() } // All channels are empty now,have a new state
30    if btp.nonEmpty then
31       remove some ⟨s,stmt⟩ from btp; restore(s); execute stmt
32    else
33       // No more backtrack points
34       send res to state maintenance
35       let s = set of states returned from state maintenance
36       restore(s)
37       unfreeze all external channels and put their data into subscribed channels
38       freeze all external channels
39    end // if
40 end // while
```