Towards Complex Actions for Complex Event Processing

Steffen Hausmann
Institute for Informatics
University of Munich
hausmann@lmu.de

François Bry
Institute for Informatics
University of Munich
bry@lmu.de

ABSTRACT

Complex actions are a natural extension for complex event processing languages needed by many applications like emergency management. In particular interactions with external actuators that are common in those applications pose challenges that need to be adequately covered. Many approaches towards actions and reactivity in event processing are, however, either too simple or too formal to model complex composite actions in a convenient manner or require a complete knowledge of the actions and of their effects.

This article proposes a pragmatic yet generic approach to complex actions in event processing which adapts to the heterogeneous and incomplete nature of physical actions. The article furthermore introduces a static semantic analysis for rejecting incorrect and undesirable programs which scales with the available information without requiring an a priori, or complete, knowledge of the actions and their consequences. The article finally describes a transformation of complex actions into complex events queries making it rather simple to add complex actions to a wide range of event processing languages.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

Keywords

Composite Actions, External Actions, Temporal Analysis, Semantic Analysis, Complex Event Processing

1. INTRODUCTION

Many applications are conveniently implemented using complex event processing techniques [19]. However, many implementations focus merely on the deduction of high-level knowledge in terms of complex events and delegate the execution of reactions, if any at all are modeled within the event processing system, to proprietary often hard coded systems. A large number of applications can substantially benefit from a new generation of event processing systems that integrate the definition of complex events with the capability of modeling the logic of reactions whereas only the execution of basic actions is delegated to external actuators.

One particular field that benefits from such a new kind of systems is emergency management in public infrastructures like subway systems and airports. Nowadays, such infrastructures are operated by humans from a central control room. Composite reactions are executed by isolated and proprietary subsystems with an incomplete knowledge and the information provided by sensor is poorly processed. Moreover, incidents in the past have shown that static procedures and human misinterpretation may result in severe casualties or damages [13, 22].

Emergency management calls for complex event processing with complex actions and fast computable simulations [4] making it possible to derive a more abstract and high level interpretation of the arriving data and to execute composite reactions that are requested by human operators.

Challenges. Although the combination of complex events and reactive rules has already been extensively investigated in the literature, the physical nature of basic actions that are eventually executed by external actuators introduces new aspects that need to be considered to obtain an adequate and effective approach suited, for instance, to emergency management applications as they are described in [25, 18].

Example 1. Smoke is the most dangerous threat to passengers and personnel during a fire in a metro station. Accordingly it is crucial to keep evacuation routes free of smoke as long as possible. This is usually achieved by adapting the ventilation regime so that a flow of fresh air keeps the smoke away from important areas.

To this end, smoke dampers are opened and ventilators are activated to generate the desired air flow that pushes the smoke out of the station. Moreover, warning signals are activated close to the outlet of the ventilation system some time before the leakage of smoke.

From this example we derive the following observations:

Physical Actions Atomic actions are executed by external actuators that interfere with the real world the underlying physical effects of which can hardly be formalised. To estimate, e.g., the effect of an airflow on the distribution

of smoke within an area, complex numeric simulations of physical effects are required [4].

Irreversible Effect Physical actions can often hardly be reversed or compensated as the caused physical effects cannot be easily undone. Once activated, the ventilators can indeed be turned off again, however, the caused airflow has already scattered the smoke what cannot be simply undone by reversing the airflow.

Timing of Actions In contrast to mere sequences of actions that are commonly used in imperative procedures, the exact timing between physical action is often crucial to obtain a desired effect. To be effective, the warning of leaking smoke needs to be issued, e.g., 20 seconds ahead of time and not just in the moment the smoke actually passes out.

Indirect Feedback External actions may fail to achieve their intended goal. However, feedback on their success usually cannot be inferred from the feedback that is provided by the contributing actuators. The adaptation of the ventilation regime is successful when the smoke actually disappeared, not when the dampers and ventilators were activated successfully.

Requirements. Based on these observations we derive the following requirements for complex actions suitable for modern and effective emergency management:

High-Level Language Emergency management has a natural need for expressiveness and ease of use. Complex actions need to be capable of modelling composite workflows in a manner that is convenient and appropriate for humans. Furthermore, complex actions must be tailored to the particularities of physical actions as they are desirable for interactions with external actuators in the infrastructure. At the same time, approaches based on intelligently acting autonomous systems cannot be used. Emergency managers are in charge and need to be in control and consequently they only accept actions that are specified in a deterministic and comprehensible way.

Integration of Events and Actions Instead of dedicating the execution of composite actions to proprietary and specialized systems without an expressive notion of complex events, it seems more appropriate to uniformly integrate the execution of composite actions into the event processing system. Complex event queries are capable of combining the information from various sources to obtain an abstract representation of the infrastructure and its condition that is valuable to control the executed procedures during runtime and to determine, e.g., the result of complex actions.

Expressive Temporal Dependencies Complex actions usually try to achieve a higher level goal that cannot be realised by individual actions but requires the collaboration of multiple actions. To this end, complex actions have a need for temporal dependencies that specify the timing and sequence for several actions in a manner that exceeds the capabilities of ordinary sequences and cases from imperative programming languages. To be valuable, the system must be able to actually execute complex actions according to their temporal dependencies.

Static Semantic Analysis Complex actions require a versatile semantic analysis that identifies errors at compile time which would otherwise only manifest during runtime. As the knowledge on runtime properties of heterogeneous actions is often incomplete, the analysis needs to scale with the available information. Basic properties of complex actions must be verifiable without specific knowledge whereas more specific properties can be verified if the corresponding knowledge is available.

Contributions. We make the following contributions:

- Identification of orthogonal dimensions that must be supported by expressive complex actions and discussion of limitations that are inherent to external actions.
- Introduction of expressive complex actions tailored to external actions with a clear separation of orthogonal aspects of action execution.
- Discussion of viable temporal dependencies between actions and elaboration of an execution strategy that satisfies those dependencies during runtime.
- Elaboration of a semantics of actions that enables a semantic analysis which ensures crucial properties of complex actions at compile time.
- Suggestion of a transformation scheme that converts complex actions to event queries which can be evaluated on top of a conventional event processing system.

2. FOUNDATIONS OF COMPLEX ACTIONS

2.1 Times Associated with Actions

The underlying time model is a crucial aspects for complex event processing that substantially influences the semantics of complex events. In event processing systems application time is often used in favour of system time to avoid unintuitive effects. In a similar way, the employed time model significantly impacts the semantics of actions and thus similar issues need to be accounted to obtain an appropriate time model for actions.

In general it seems desirable to apply a synchronous time model and that the times associated with an action instance are determined by the corresponding actuator that actually executes it. In this way, the impact of latency and network delays is minimized and in consequence the semantics of actions becomes more meaningful and stable.

We distinguish three different times of actions: one related to the beginning and two related to the ending of an action.

Initiation Time. The initiation time of an action refers to the time the action is deemed to begin.

Success and Failure Times. The end of an action is denoted by the time of its success or failure. Naturally, each action can only either succeed or fail. In the context of emergency management, actions are furthermore considered as failed if they did not succeed within an application dependent amount of time.

Note that these times are subject to a priory unknown runtime effects and cannot be directly influenced. In particular the initiation times of actions can only be affected indirectly: Actions are requested by the event processing system as soon as the premises of the corresponding reactive rule are satisfied. However, this constraints only the earliest possible times at which actions can be conceptually initiated whereas the times of their actual initiation are subject to runtime effects, such as, latency and network delay.

This aspect substantially impacts the way how actions can be executed during runtime and introduces limitations on viable temporal relations that can be actually guaranteed during runtime.

2.2 Indirect Feedback on Physical Actions

Physical actions are requested within the event processing system and executed by external actuators. As a consequence the system has inherently no general knowledge on the progress of requested actions. Instead it depends on the feedback that is provided by the corresponding actuators to determine the current status of running actions. However, due to the heterogeneity of actuators in large infrastructures, the quality of feedback may vary substantially. For instance, not every actuator can provide the feedback that is desired, to determine when and if an action was actually successful, some even cannot provide any feedback at all.

As a consequence, it is often mandatory to rely on indirect feedback from related sensors that allows inferences on the execution status of actions. For instance, to determine whether a ventilator was activated successfully, one can use the information on the current airflow measured by an adjacent anemometer. If even no indirect feedback is available, domain knowledge can be used to specify, for instance, that the activation of the ventilator is successful 20 seconds after the request was emitted by the event processing system.

In summary, it is desirable to obtain the feedback on the execution of actions directly from the actuator, but due to technical limitations the feedback may need to be inferred from other sources including very generic information provided by the event processing system itself. As a consequence, the times associated with external actions need to be adjustable according to the capabilities of the actuators and the requirement of the programmer as there is no suitable default that equally fits all kinds of diverse actions.

2.3 Dimensions of Complex Actions

A language towards complex actions for emergency management must support (at least) the three complementary dimensions of action composition, temporal dependencies, and execution result. A fourth dimension, temporal assertions, seems desirable to increase the quality and robustness of programs but it does not result in a higher expressiveness.

Action composition. Complex actions are composed from several (atomic or complex) sub-actions that are executed in combination to achieve a certain higher level goal that cannot be achieved by single and more basic actions. Naturally, a language for complex actions must support the composition of several actions into one composite complex action.

Note that emergency management requires temporal dependencies that exceed the expressiveness of common sequences and choice as they are available in imperative languages. However, action composition covers only the mere collection of several actions whereas monolytic operators available in other approaches, like sequences, are expressed by means of more generic and expressive temporal dependencies between actions.

Temporal dependencies. Temporal relations are mandatory to specify the timing and execution order of actions which must be satisfied when the action is executed. Commonly used temporal dependencies between actions are for instance *before* and *after* which may additionally specify an optional duration (e.g., *20 seconds after*).

To obtain expressive complex actions, it is crucial that multiple dependencies can be independently specified for the same action. Note that this requirement is often not satisfied by monolytic composition operators which interleave temporal relations and action composition.

To be suitable for external actions with uncertain results, temporal relations must furthermore discriminate between success and failure of actions to allow different reactions based on the result of preceding actions (e.g., *only execute a after b was successful*).

Execution result. Complex actions need a mean to specify whether their intended effect has been accomplished or not, that is, if their execution was successful or failed. Without a notion of success and failure of actions, composite workflows that invoke different alternatives to adapt to the result of preceding actions cannot be easily modeled.

As the desired effect of actions may not only depend on the success of their sub-actions, those means must be expressive enough to specify generic event patterns that verify the success and failure of the desired effects of the complex action.

Temporal assertions. Temporal assertions specify temporal conditions between actions that need to be satisfied during runtime. In contrast to temporal dependencies, these conditions have no effect on the execution of the composite action. It is just verified during compile time that the conditions *will* be satisfied if the action is executed according to its temporal dependencies.

It is desirable that the verification of assertions does not rely on comprehensive domain knowledge on actions, as the available knowledge is in practice often incomplete. However, if specific domain knowledge is actually available for certain actions, it should be considered by the analysis to obtain stronger results.

Temporal assertions do not increase the expressiveness of actions, but they facilitate the development of more robust code as they they specify conditions on the behaviour during runtime that are verified by the system at compile time.

Besides the capabilities of four dimensions, a clear separation of concerns of the orthogonal properties of the dimensions seems desirable. The benefits of a clear separation of concerns that are widely recognised for rule based languages, in particular for rule based event query languages [7]. These benefits can be naturally generalised for complex actions in complex event processing. Without a clear separation of orthogonal concepts, complex actions lose expressiveness and become cumbersome and unintuitive to write. This effect increases the more dimensions are considered for the actions.

2.4 Semantic Analysis for Actions

The clear separation of concerns and a good coverage of the four orthogonal dimensions of actions is arguably desirable for complex actions, in particular for complex actions in emergency management. However, the resulting expressiveness comes at a price, namely the need for a strong semantic analysis capable of rejecting incorrect and faulty programs. The semantic analysis of complex actions should preferably cover at least the three following aspects.

Viability of Temporal Dependencies. Atomic temporal dependencies need to be viable in the sense that they can be actually satisfied by the system during runtime. As the referred time-points of actions can only be affected indirectly, arbitrary temporal dependencies between actions may not necessarily be satisfiable during runtime.

For instance, due to inherent runtime effects, it is impossible for the system to guarantee that two actions are actually initiated at the same time, because the distribution of the action request to the actuator is subject to latency. Note that this still holds if the initiation refers to the time the request is emitted by the system.

Fairness of Actions. Using action identifiers and generic relations to specify temporal dependencies between actions seems desirable. In fact, it is even mandatory to facilitate the integration of multiple independent dimensions without losing expressiveness.

However, the flexibility of temporal dependencies may result in inconsistent specifications, for instance in cyclic dependencies between actions, that prevent sub-actions from being executed. Accordingly, the semantic analysis must verify at compile time whether the temporal dependencies allow that all sub-actions can actually be executed during runtime and that there is no "dead code" that contains actions which will never be executed.

Entailment of Assertions. Specifying temporal assertions is only meaningful if it is verified during compile time that the corresponding assertions of an action will actually be satisfied during runtime.

However, the verification often requires domain knowledge that might not be available for all kinds of physical actions. Accordingly, it is mandatory that the semantic analysis can incorporate such domain knowledge, specified, e.g., in the schema of actions, but does not rely on it to work at all.

3. ACTIONS IN A HIGH-LEVEL LANGUAGE

In the following, we will elaborate complex actions that can be integrated into high-level event query languages. To this end, a short overview of the event query language Dura is given which has been introduced in [16] and which is exemplarily used as a basis for our work. Subsequently, we will extend the event query language with expressive complex actions that aim at a full coverage of the aforementioned dimensions of complex actions.

3.1 Event Processing with Dura in a Nutshell

Dura is a high-level rule based complex event processing language in the spirit of the XML query and transformation language Xcerpt [9] and the rule based event query language XChange^{EQ} [6].

Event queries in Dura are characterized by a pattern based query approach, versatile temporal dependencies between events, versatile negation and grouping capabilities, and a clear separation of query dimensions that is desirable to obtain a high expressiveness of the language [7]. Moreover, Dura comes with stateful objects that represent non-volatile data which can be updated in non-destructive and declarative fashion and support of multiple external time models.

Atomic Event Queries. Events are represented as structured data, similar to structs known from `C`. Every event has a name, contains a unique identifier and further user defined attributes in its payload, and is associated with a time interval.

Events are queried by means of a pattern based approach, that is, the query pattern resembles the data of the event and variables are specified in the pattern where data should be extracted. In addition, an event identifier is introduced that precedes the query that is used in composite queries to refer, e.g., to the time of the matched event.

```
event e: smoke{ area{var Area}, amount{var C} }
```

The preceding query matches `smoke` events and binds the value of the `area` and `amount` attributes to variables. Note that either values or composite data can be bound to variables and that the query pattern may be incomplete, omitting irrelevant attributes.

Composite Event Queries. Several event queries are combined by means of the operators `and`, `or`, and `not`. In addition to the mere composition of queries, temporal and other dependencies between events and the data they carry are given in a separate `where` part that is appended to the query.

```
and{
  event e: smoke{ area{var Area}, amount{var C} },
  event f: temp{ area{var Area}, value{var T} }
} where { {e,f} within 2 min, C > 0.1, T > 50 }
```

This query matches `smoke` and `temp` events that occur within 2 minutes in the same area, note the implicit join over the variable `Area`, and which report a temperature above 50°C and a smoke concentration that exceeds 10 percent.

Event queries are purely declarative and do not consume or absorb any events. Accordingly, the same event can be matched by a rule multiple times. For instance, a smoke event may be matched twice if two suited temp events occur within the appropriate amount of time.

Deductive Rules. Deductive rules derive higher level events based on the occurrence of events in the stream. They correspond to materialized views from database systems.

```
DETECT
  fire{ var Area }
ON
  and{
    event e: smoke{ area{var Area}, amount{var C} },
    event f: temp{ area{var Area}, value{var T} }
  } where { {e,f} within 2 min, C > 0.1, T > 50 }
END
```

In their head deductive rules contain a data term with variables which are replaced by the values obtained during the evaluation of the query in the body of the rule. Note that the values for the unique identifier and the time of events are automatically determined by the system.

The given rule derives new `fire` events carrying the originating area of the fire in their payload whenever the query from above matches the stream of events.

3.2 Complex Actions for Dura

Complex actions aim at a high expressiveness with a full coverage of all four dimensions of actions. In the following, complex actions for Dura are introduced in the context of an emergency management related scenario that resembles the one given in the introduction.

Atomic Actions. Atomic actions are specified in a manner similar to the specification of atomic event queries.

```
action a: adapt-ventilation{ var Area }
```

However, instead of extracting values from the pattern, as in case of event queries, the values that are already bound to variables are injected to the corresponding actions as parameters.

Reactive Rules. Reactive rules are the counterpart of deductive rules. They trigger the execution of actions as a

reaction to the occurrence of events. Note that there is no automatic conflict resolution for reactive rules matching the same events in Dura. Accordingly, all reactive rules matching the stream of events are always triggered. Conflicts between rules are explicitly resolved by adding further conditions to their event queries.

```
ON
  event e: fire{ var Area }
DO
  action a: adapt-ventilation{ var Area }
END
```

For the sake of simplicity, we assume that an enterprise service bus [10] is available and that external actuators are connected to the bus by means of appropriate adaptors.

Action Composition. Composition of complex actions is expressed in a manner similar to the composition of event queries. Several actions are grouped together by means of the `compound` operator and identifiers are introduced that refer to the actions they precede.

```
compound{
  action a: open-fire-dampers{ var Area },
  action b: activate-ventilators{ var Area }
}
```

Note that, due to the clear separation of orthogonal dimensions, `compound` is the only available operator that is required for the composition of actions. It just specifies the actions that are executed in combination with their concrete parameters, but does not describe further dependencies between actions. So in this case, both actions are simply executed concurrently.

Temporal Dependencies. Temporal dependencies between actions are specified in the `where` part of complex actions. They refer to the actions from the separate `compound` part by means of the action identifiers and specify the timing of actions. To this end, action identifiers are used in combination with *init*, *succ*, and *fail* to distinguish between the different time-points associated with the actions.

Temporal dependencies are specified by means of conjunctions of inequalities that determine lower bounds for the initiation of actions. For other applications it seems appropriate to furthermore support disjunctions of temporal dependencies that enable non-deterministic actions. However, for emergency management applications the determinism of actions is a crucial requirement and as a consequence, disjunctive dependencies are not further considered here although they can be integrated in our approach. An extension integrating disjunctive dependencies is discussed in Sec. 7.

```
compound{
  action a: open-fire-dampers{ var Area },
  action b: activate-ventilators{ var Area }
} where { succ(a) <= init(b) }
```

The preceding complex action uses temporal dependencies to specify that the ventilators should only be activated after the fire dampers have been successfully opened. Note that this behavior corresponds to a simple sequence of actions that can be also specified by many other approaches that support composite actions. However, due to the clear separation of dimensions more elaborated dependencies can be easily specified whereas approaches that interleave the composition of actions and their temporal dependencies in monolithic operators fail to express the following extension of the preceding example.

```
compound{
  action a: open-fire-dampers{ var Area },
  action b: activate-ventilators{ var Area },
  action c: warn-of-smoke-emission{ var Area }
} where { succ(a) <= init(b),
          init(c) + 20 sec <= init(b) }
```

The complex action is complemented by a third action that warns persons close to the outlet of the ventilation system of the imminent emission of smoke. To be effective, a time delay of 20 seconds between the issue of the warning and the actual emission of smoke is added to the temporal dependencies of the action.

Execution Result. Complex actions usually try to achieve a higher level goal that cannot be realised by individual actions. Naturally, the success of the action depends on the achievement of this goal which often cannot be inferred from the raw success of the comprising actions. As a consequence, Dura employs versatile event queries to specify the success of actions beyond the success of their sub-actions.

The success and failure of complex actions is specified in the dedicated `succeeds on` part by means of common event queries. The failure of actions is implicitly specified, as in emergency management actions are deemed as failed if they are not successful within a certain amount of time. The `where` part of the event query specifying the success of the action may as well refer to the time-points of sub-actions by means of their action identifiers.

Similar to the specification of the success of actions, the initiation of external non-composite actions can be specified by means of an `initiated on` part that is incorporated to the (schema) specification of the atomic action.

```
compound{
  action a: open-fire-dampers{ var Area },
  action b: activate-ventilators{ var Area }
} where { succ(a) <= init(b) }
  succeeds on {
    event e: smoke{ area{var Area}, amount{var C} }
      where { C < 0.2, end(e)-init(a) < 60 sec }
  }
```

In this case, the action that is intended to extract smoke from a certain area is deemed successful if the smoke concentration drops below 20% in the respective area within one minute from the beginning of the action and fails otherwise. Note that the `where` part of the complex action and of the query in the `succeeds on` part are both referring to the `open-fire-damper` action by means of the identifier `a`.

Temporal Assertions. Temporal assertions are specified in the `hence` part of complex actions. They are denoted in a way that resembles the specification of temporal dependencies but are in general more expressive, as arbitrary combinations of conjunctions and disjunctions of inequalities can be specified. Temporal assertions serve to express additional formulas that must hold during runtime. Assertions that do not necessarily hold during runtime are detected during compile time and result in compilation errors.

Note that domain knowledge, such as, the maximal duration of the corresponding actions and the latency of the system, may be required to verify assertions during compile time. The available domain knowledge is specified in the schema of actions by means of inequalities similar to those of assertions. However, if no additional domain knowledge is available or if it is omitted from the schema, assertions may

be falsely rejected due although they are actually always satisfied during runtime.

```
compound{
  action a: open-fire-dampers{ var Area },
  action b: activate-ventilators{ var Area }
} where { succ(a) <= init(b) }
  hence or{ succ(b)-init(a) <= 20 sec,
            fail(b)-init(a) <= 20 sec }
```

Assertions are used in this case to ensure that the sequential execution of `open-fire-dampers` and `activate-ventilators` is completed (regardless of its result) within 20 seconds. Recall that assertions do not influence the execution of actions and are just verified during compile time. To actually verify this particular assertion, reliable information on the duration of both actions and the latency needs to be available in the schema of the actions.

Complex Action Rules. Complex action rules assign names to (anonymous) complex actions in a way that resembles procedures that assign names to certain fragments of code.

```
FOR
  adapt-ventilation{ var Area }
DO
  compound{
    action a: open-fire-dampers{ var Area },
    action b: activate-ventilators{ var Area }
  } where { succ(a) <= init(b) }
    succeeds on {
      event e: smoke{ area{var Area}, amount{var C} }
        where { C < 0.2, end(e)-init(a) < 60 sec }
    }
END
```

Accordingly, the specified complex action can be executed by referring to its name `adapt-ventilation` instead of repeating the entire code of the body of the given rule.

3.3 Satisfying Temporal Dependencies

During runtime complex actions are actually executed by the event processing system according to their temporal dependencies. To this end, the system can defer actions whose initiation is explicitly specified in the temporal dependencies of the complex action.

To obtain clear and reasonable semantics, the system may not implicitly assume dependencies that are not explicitly specified by the user. In general, implicit assumptions are avoided in our approach as they influence the semantics of actions in a way that can easily be overlooked by programmers. Note that this is a major difference to approaches concerned with the planning and scheduling of actions [31].

To satisfy the following temporal dependency, the system simply defers the initiation of `b` until the success of `a` has been observed. If several dependencies for the initiation of an action are given, the system simply defers its initiation until all of them are satisfied.

```
where { succ(a) <= init(b) }
```

In contrast, the subsequent temporal dependency is invalid as it does not constrain the initiation of actions and can hence only be observed during runtime but cannot be satisfied in general without considering further dependencies.

```
where { succ(a) <= succ(b) }
```

In summary, temporal dependencies specify lower bounds for the initiation of actions that need to be exceeded before the action is requested by the system. Although the available constraints seem rather limited, we will determine that due to the inherent properties of external actions other dependencies cannot be satisfied in general.

The execution of complex actions can thus be understood as some kind of feedback loop. The event processing system requests the execution of actions which eventually results in the observation of their success or failure. This, in turn, determines the lower bounds for the initiation of further actions which will be eventually exceeded. Eventually these actions will be requested for execution and so forth.

4. SEMANTICS OF COMPLEX ACTIONS

The execution strategy of complex actions combined with the loose structure of temporal dependencies bears some pitfalls. For instance, cyclic dependencies can be specified that prevent some, or even all, actions from being executed during runtime. To overcome those undesirable effects while maintaining the expressiveness of the temporal constraints we elaborate a semantic analysis that is capable of detecting such situations during compile time.

However, to obtain a meaningful analysis, the correctness of an algorithm for the semantic analysis of actions must be verified formally. To this end, a formal semantics of complex actions is required that is generic enough to model physical action but that is at the same time specific enough to formally prove the desired properties.

4.1 Formalization of Complex Actions

For convenience, complex actions are formalized in a more concise manner that omits the verbose syntactic constructs of the language and contains only information that is crucial for the intended analysis.

Definition 1. The set of *variables* is denoted $\mathcal{V} = \mathcal{V}_o \cup \mathcal{V}_a$. For each action identifier f, the variable $f_{\text{init}} \in \mathcal{V}_a$ is called *affected* variable and the variables $f_{\text{succ}}, f_{\text{fail}} \in \mathcal{V}_o$ are called *observed* variables.

Variables correspond to the times associated with actions and accordingly their values are determined by external components. However, the initiation of actions can be deferred by the system and thus the system can determine lower bounds for the value of affected variables whereas the values of observed variables can indeed just be observed.

Note that these notions resemble activated and received time-points from [31].

Definition 2. The set of *atomic temporal dependencies* is a set of triples denoted $\mathcal{C} = \mathcal{V} \times \mathbb{Q} \times \mathcal{V}$.

Informally these triples correspond to temporal dependencies of complex actions. For instance, $(a_{\text{succ}}, 0, b_{\text{init}}) \in \mathcal{C}$ corresponds to the dependency `succ(a) <= init(b)` from above. For convenience, $(u, d, u'), (v, -d, v'), (w, 0, w') \in \mathcal{C}$ can also be written $u \dot{+} d \dot{\leq} u'$, $v \dot{-} d \dot{\leq} v'$ and $w \dot{\leq} w'$. Other relations, like $\dot{<}$ and $\dot{=}$, are not considered here, but note that they can be expressed by means of the given ones.

Definition 3. A *complex action* C is formally represented by a conjunction of temporal dependencies $C = \bigwedge_i d_i$ with $d_i \in \mathcal{C}$.

Definition 4. The *assertions* H of complex actions are represented by conjunctions of disjunctions of temporal dependencies $H = \bigwedge_i \bigvee_j d_{ij}$ with $d_{ij} \in \mathcal{C}$.

Definition 5. The *domain knowledge* D on actions is obtained from their schema and represented just like their assertions are by $D = \bigwedge_i \bigvee_j d_{ij}$ with $d_{ij} \in \mathcal{C}$.

For convenience, a conjunction $C = \bigwedge_i d_i$ is represented by a set $C = \bigcup_i d_i$. As the number of dependencies is always finite, both representations are used interchangeably. Moreover, complex actions from Dura are formally identified by their corresponding set of temporal dependencies.

Definition 6. The *variables* of a complex action $C \subseteq \mathcal{C}$ are denoted

$$\mathrm{var}(C) = \left\{ v \;\middle|\; (v \dot{+} d \dot{\leq} v') \in C \;\vee\; (v' \dot{+} d \dot{\leq} v) \in C \right\}$$

Definition 7. The *axiomatic closure* of a complex action $C \subseteq \mathcal{C}$ is a set $C_{\mathcal{A}} \supseteq C$ that contains the following implicit axioms on sub-actions

$$C_{\mathcal{A}} = C \;\cup \bigcup_{f_{\text{init}} \in \mathrm{var}(C)} \left\{ \bot_{\text{init}} \dot{\leq} f_{\text{init}},\; f_{\text{init}} \dot{\leq} f_{\text{succ}},\; f_{\text{init}} \dot{\leq} f_{\text{fail}} \right\}$$

whereby the special variable $\bot_{\text{init}} \in \mathcal{V}_o$ refers to the initiation of the complex action.

The additional dependencies that are introduced by the axiomatic closure ensure that no sub-action is initiated before the complex action has been initiated and that the success and failure of sub-actions occur after their initiation.

Example 2. The complex action `adapt-ventilation` from Sec. 3.2 is formally represented by the set $C = \{a_{\text{succ}} \dot{\leq} b_{\text{init}}\}$ with the axiomatic closure

$$C_{\mathcal{A}} = C \cup \left\{ \bot_{\text{init}} \dot{\leq} a_{\text{init}},\; a_{\text{init}} \dot{\leq} a_{\text{succ}},\; a_{\text{init}} \dot{\leq} a_{\text{fail}}, \right. \\ \left. \bot_{\text{init}} \dot{\leq} b_{\text{init}},\; b_{\text{init}} \dot{\leq} b_{\text{succ}},\; b_{\text{init}} \dot{\leq} b_{\text{fail}} \right\}$$

The following domain knowledge on `open-fire-damper` actions limits their duration to 5 seconds.

$$D = \left\{ a_{\text{succ}} \dot{-} 5 \dot{\leq} a_{\text{init}} \vee a_{\text{fail}} \dot{-} 5 \dot{\leq} a_{\text{init}} \right\}$$

Note that the system latency can be specified in a similar manner by constraining the time between the initiation of actions and the end of their predecessors.

4.2 Formalizing Viable Temporal Dependencies

By design, complex actions in Dura only support temporal dependencies in their `where` part that specify lower bounds on the initiation of actions. However, this limitation is actually not specific for Dura but applies in general if the initiation of actions can only be indirectly affected.

To satisfy atomic temporal dependencies during runtime, there must be lower bounds for their affected variables so that the temporal dependencies are satisfied for all valid values of their observed variables. In the following we discriminate between *viable* dependencies for which proper lower bounds exist and *observable* dependencies that may not be satisfied in all situations.

There are four different categories of atomic dependencies. Note that atomic dependency with variables that refer to the same action, e.g., $a_{\text{succ}} \dot{-} 5 \dot{\leq} a_{\text{init}}$, are not considered in the following, because they are always observable as the duration of actions cannot be influenced by the system.

$(v_1 \dot{+} d \dot{\leq} v_2) \in \mathcal{V}_o \times \mathbb{Q} \times \mathcal{V}_o$: The variables v_1 and v_2 are both observed variables and hence the system has no direct influence on their values. Accordingly, those kinds of dependencies are observable dependencies.

$(v_1 \dot{+} d \leq v_2) \in \mathcal{V}_o \times \mathbb{Q} \times \mathcal{V}_a$: The value $\ell_2 = v_1 \dot{+} d$ is a lower bound for the affected variable v_2 so that the formula is satisfied for all values of the observed variable v_1. Accordingly those kinds of dependencies are viable dependencies. However, in practice, only $d \in \mathbb{Q}^+$ is reasonable, as the value of v_1 is not determined until it is exceeded.

$(v_1 \dot{+} d \dot{\leq} v_2) \in \mathcal{V}_a \times \mathbb{Q} \times \mathcal{V}_o$: There is no lower bound ℓ_1 for the affected variable v_1 that implies the formula (the assumption there is a bound ℓ_1 leads to a contradiction for $v_2 < \ell_1 + d$). Therefore, those kinds of dependencies are observed dependencies.

$(v_1 \dot{+} d \dot{\leq} v_2) \in \mathcal{V}_a \times \mathbb{Q} \times \mathcal{V}_a$: In general, there are no lower bounds ℓ_1 and ℓ_2 for v_1 and v_2 that imply the formula (the assumption there are such bounds leads to a contradiction for $v_2 \dot{<} v_1 \dot{+} d$). Accordingly, those kinds of dependencies are observed dependencies.

In summary, the set $\mathcal{V}_o \times \mathbb{Q}^+ \times \mathcal{V}_a$ exactly corresponds to the set of viable dependencies which can be satisfied during runtime by choosing the right lower bounds for their affected variable. Naturally, viable dependencies are specified in the `where` part of actions whereas observable dependencies are specified in their `hence` part where it is verified by the semantic analysis that they actually hold during runtime.

4.3 Semantics of Complex Actions

In order to prove properties of complex actions, their behaviour must be characterized in a formally precise manner. To this end, we develop a notion of runtime traces that formalizes the behavior of complex actions during runtime.

Definition 8. $\mathbb{P} = \mathbb{Q}^+ \cup \{\infty\}$ and $\mathbb{D} = \mathbb{Q}^+ \cup \{\infty\}$ denote the set of *time-points* and the set of *durations*. Time-points and durations can be added up in a canonical manner

$$+ : \mathbb{P} \times \mathbb{D} \to \mathbb{P},\; (p, d) \mapsto (p + d).$$

The terms are discriminated to emphasise their different semantics. Durations will be used to describe possible delays during runtime whereas time-points will refer to the absolute times, for instance, at which time an action succeeded.

Definition 9. A *variable assignment* maps variables to values in $\mathbb{D}$ or $\mathbb{P}$. In the following, variable assignments are denoted by sets of variable value pairs as it is known from substitutions used in model theory [2].

Variable assignments can be naturally applied to composite syntactical expressions. For $v_i \in \mathcal{V}$ and $d_i \in \mathbb{Q}^+$

$$\tau(v_1 \dot{+} d_1) = \tau(v_1) + d_1$$
$$\tau\left(\{v_1 \dot{+} d_1, \ldots, v_k \dot{+} d_k\}\right) = \left\{\tau(v_1 \dot{+} d_1), \ldots, \tau(v_k \dot{+} d_k)\right\}$$

Definition 10. Given a complex action $C \subseteq \mathcal{C}$, a variable assignment $\Delta : \mathrm{var}(C) \to \mathbb{P}$ is called a *trace* of C.

Traces are intended to describe the execution of actions. To this end, a trace maps variables corresponding to the initiation, success and failure of sub-actions to actual time-points. Thereby the value $\infty \in \mathbb{P}$ indicates that an action has not been initiated, did not succeed, and did not fail.

Obviously not every arbitrary trace corresponds to how the action is actually executed during runtime. Some sub-actions may, according to the trace, be successful before

they begin, the temporal dependencies of the complex action may not be satisfied by the trace, etc. To obtain a more appropriate representation of complex actions, traces need to incorporate the temporal dependencies between actions and the execution strategy for actions.

However, in general, it cannot be known in advance how long it will take to execute an external action and whether the execution will be successful or not. Therefore these runtime effects are abstracted away by means of so-called scenarios.

Definition 11. Given a complex action $C \subseteq \mathcal{C}$, a variable assignment $\Delta : \mathrm{var}(C) \to \mathbb{D}$ is called a *scenario* of C. For convenience $\Delta(v)$ is also denoted Δ_v.

Each scenario describes one particular series of developments for the different outcomes of sub-actions and time delays that can potentially occur during runtime.

Example 3. The scenario $\delta \supseteq \left\{7/\bot_{\mathrm{init}}, 1/a_{\mathrm{init}}, 5/a_{\mathrm{succ}}, \infty/a_{\mathrm{fail}}\right\}$ describes, e.g., that the action a is initiated 1 ms after it has been requested and succeeds 5 ms after it has been initiated.

Definition 12. For a complex action C the *preconditions* for the determination of a variable $v \in \mathrm{var}(C)$ are denoted

$$\mathrm{pre}_C(v) = \left\{ v_i \dot{+} d_i \;\middle|\; (v_i \dot{+} d_i \dot{\leq} v) \in C \right\}$$

Informally, the preconditions of a variable f_{init} is the set of lower bounds which must each be exceeded before the action f can be requested by the event processing system. Therefore, the time-point when the action is actually initiated depends on the largest of those bounds and the latency between the request and the initiation which is available from the considered scenario.

Definition 13. Given a complex action $C \subseteq \mathcal{C}$ and a runtime scenario Δ. Then the operator $T_{\Delta,C}$ maps variable assignments to variable assignments with

$$T_{\Delta,C}(\sigma) = \left\{ \left(\max\left\{\sigma\big(\mathrm{pre}_C(v)\big)\right\} + \Delta_v \right) \Big/ v \;\middle|\; \forall v' \in \mathrm{var}\big(\mathrm{pre}_C(v)\big) : v' \in \mathrm{dom}(\sigma) \right\}$$

Hereby $\max \emptyset = 0 \in \mathbb{P}$ and thus the operator $T_{\Delta,C}$ is actually a mapping between (incomplete) traces.

T formalizes on step of the feedback loop described in Sec. 3.3. It takes as an argument an incomplete trace that contains the values of variables that have already been observed and adds values for the initiation, success and failure of actions in response to the given (observed) values in compliance with the scenario Δ. To obtain a complete trace, the operator is applied multiple times to incrementally determine values for all variables of the action C.

Note that the operator T makes a transition from merely syntactic formulas on the right to actual time-points that are assigned to v on the left.

Definition 14. The *powers* of T are inductively defined by

$$T^0 = \emptyset,\ T^{n+1} = T\left(T^n\right)$$

and the *least fixpoint* of T is denoted $\mathbf{T}$.

As the operator T is monotonic [2], it has a unique least fixpoint. Moreover, the fixpoint is reached after a finite number of iterations. For a formal proof refer to the appendix in the electronic version of this paper [17].

Basically, the fixpoint $\mathbf{T}$ describes, based on one particular scenario, how the action will be executed by the system if the given delays are actually observed during runtime.

Example 4. A complete iteration of T is given in the following example. Note that not only the initiation of actions, but also the time-point of their success and failure are determined by T according to the scenario Δ. For $C = \left\{a_{\mathrm{succ}} \dot{+} 5 \;\dot{\leq}\; b_{\mathrm{init}}\right\}$ and with $T = T_{\Delta,C_{\mathcal{A}}}$

$$\begin{aligned}
T^1 &= \left\{\Delta_{\bot_{\mathrm{init}}} \big/ \bot_{\mathrm{init}}\right\} \\
T^2 &= T^1 \cup \left\{\Delta_{\bot_{\mathrm{init}}} + \Delta_{a_{\mathrm{init}}} \big/ a_{\mathrm{init}}\right\} \\
T^3 &= T^2 \cup \left\{\Delta_{\bot_{\mathrm{init}}} + \Delta_{a_{\mathrm{init}}} + \Delta_{a_{\mathrm{succ}}} \big/ a_{\mathrm{succ}}\right\} \\
&\quad \cup \left\{\Delta_{\bot_{\mathrm{init}}} + \Delta_{a_{\mathrm{init}}} + \Delta_{a_{\mathrm{fail}}} \big/ a_{\mathrm{fail}}\right\} \\
T^4 &= T^3 \cup \left\{\Delta_{\bot_{\mathrm{init}}} + \Delta_{a_{\mathrm{init}}} + \Delta_{a_{\mathrm{succ}}} + 5 + \Delta_{b_{\mathrm{init}}} \big/ b_{\mathrm{init}}\right\} \\
T^5 &= T^4 \cup \left\{\Delta_{\bot_{\mathrm{init}}} + \Delta_{a_{\mathrm{init}}} + \Delta_{a_{\mathrm{succ}}} + 5 + \Delta_{b_{\mathrm{init}}} + \Delta_{b_{\mathrm{succ}}} \big/ b_{\mathrm{succ}}\right\} \\
&\quad \cup \left\{\Delta_{\bot_{\mathrm{init}}} + \Delta_{a_{\mathrm{init}}} + \Delta_{a_{\mathrm{succ}}} + 5 + \Delta_{b_{\mathrm{init}}} + \Delta_{b_{\mathrm{fail}}} \big/ b_{\mathrm{fail}}\right\} \\
T^6 &= T^5 = \mathbf{T}
\end{aligned}$$

With the concrete scenario δ from Ex. 3 and $T = T_{\delta,C_{\mathcal{A}}}$

$$\mathbf{T}(\bot_{\mathrm{init}}) = 7, \quad \mathbf{T}(a_{\mathrm{init}}) = 8, \quad \mathbf{T}(a_{\mathrm{succ}}) = 13$$

which means that the complex action is initiated at time 7, a is initiated at time 8 and succeeds at time 13. However, by design, the fixpoint $\mathbf{T}$ may not contain values for all variables of the complex action.

Example 5. For $C = \left\{a_{\mathrm{succ}} \dot{+} 5 \dot{\leq} b_{\mathrm{init}},\ b_{\mathrm{succ}} \dot{+} 3 \dot{\leq} a_{\mathrm{init}}\right\}$ the event processing system will neither initiate a nor b during runtime as both actions depend on each other. Accordingly, runtime traces of C must provide ∞ as a time-point of a_{init} and b_{init}. However, the fixpoint of $T = T_{\Delta,C_{\mathcal{A}}}$ expresses this by containing neither of both variables instead of mapping the variables to ∞.

$$T^0 = \emptyset, \quad T^1 = \left\{\Delta_{\bot_{\mathrm{init}}} \big/ \bot_{\mathrm{init}}\right\}, \quad T^2 = T^1 = \mathbf{T}$$

This observation leads to the following definition of runtime traces, which informally maps missing values to ∞.

Definition 15. Suppose $C \subseteq \mathcal{C}$ is a complex action. A trace of C is called *runtime trace* if there is a scenario Δ such that for $T = T_{\Delta,C_{\mathcal{A}}}$ holds

$$\begin{aligned}
\tau|_{\mathrm{dom}(\mathbf{T})} &= \mathbf{T} \\
\forall v \notin \mathrm{dom}(\mathbf{T}) :\ \tau(v) &= \infty
\end{aligned}$$

and furthermore for all $f_{\mathrm{init}} \in \mathrm{dom}(\mathbf{T})$ holds

$$\mathbf{T}(f_{\mathrm{init}}) \neq \infty \implies \left(\mathbf{T}(f_{\mathrm{succ}}) \neq \infty \iff \mathbf{T}(f_{\mathrm{fail}}) = \infty\right) \quad (\star)$$

Runtime traces formally describe how complex actions are executed during runtime based on a scenario that describes the time delays that occur and determines the results of actions. They are used in the following to formally analyse the runtime properties of complex actions.

The condition $(\star)$ ensures that actions eventually either succeed or fail if they are initiated. This fundamental assumption seems very natural for actions in general and in

particular for emergency management purposes. It is thus directly incorporated into the definition of runtime traces. Note that less generic properties of specific actions can be specified in the domain knowledge associated with their schema when necessary.

5. SEMANTIC ANALYSIS OF ACTIONS

5.1 Preliminaries

Definition 16. A *disjunctive temporal problem* (DTP) [29] is a conjunction of disjunctive constraints $\bigwedge_i \bigvee_j c_{ij}$, where the c_{ij} have the form $l \leq v - v' \leq u$, v and v' represent variables that designate time-points, and $l, u \in \mathbb{Q}$.

Checking the consistency of a DTP is known to be NP-hard and there are extensions with the domain $\mathbb{Q} \cup \{\infty\}$ and with support of strict inequalities [29, 5].

In the following, the execution of complex actions that is formalized by runtime traces is expressed by means of solutions to disjunctive temporal problems. In this way, the entailment of assertions can be reduced to the inconsistency of a DTP which can be verified by established approaches.

Definition 17. A *dependency graph* is a directed weighted graph given by a triple $G = (V, E, w)$ of the set of vertices $V \subseteq \mathcal{V}$, the set of edges $E \subseteq V \times V$ and the weight function $w : E \to \mathbb{Q}$. Furthermore, the notation $v \overset{p}{\rightsquigarrow} v'$ indicates that there is a *path* p from v to v' in the graph.

A complex action C is represented by means of a dependency graph G_C by adding an edge $v \longrightarrow v'$ labeled $-d$ to the graph for each $(v' \dot{+} d \dot{\leq} v) \in C$. Note that a similar representation is used to solve simple temporal problems, a variation of DTPs, in polynomial time [12].

The graph representation of a complex action is used in the following to conveniently verify fairness properties of the action by checking properties of the graph.

Definition 18. The *canonical domain knowledge* of a complex action C is a DTP that corresponds to $(\star)$ from Def. 15. With $v \doteq \infty$ abbreviating $\bot_{\text{init}} \dot{+} \infty \dot{\leq} v$ it is denoted

$$C_{\mathcal{D}} \;=\; \bigwedge_{f_{\text{init}} \in \text{var}(C)} \Big(\big(f_{\text{succ}} \doteq \infty \;\wedge\; f_{\text{fail}} \dot{<} f_{\text{succ}} \big) \vee \big(f_{\text{fail}} \doteq \infty \;\wedge\; f_{\text{succ}} \dot{<} f_{\text{fail}} \big) \;\vee\; f_{\text{init}} \doteq \infty \Big)$$

5.2 Formal Properties of Complex Actions

For space limitations we just give the intuition behind the proofs here and refer the interested reader to the appendix in the extended electronic version of this paper [17].

Definition 19. A complex action $C \subseteq \mathcal{C}$ is *fair* iff for all $f_{\text{init}} \in \text{var}(C)$ there is a runtime trace τ with $\tau(f_{\text{init}}) \neq \infty$.

The preceding definition formalizes the notion of fairness from Sec. 2.4. Accordingly an action is fair, if for each of its sub-actions there is at least one scenario in which the sub-action is actually executed. That is, the complex action contains no "dead code" which is never executed during runtime.

THEOREM 1. *A complex action C is fair iff $G_{C_{\mathcal{A}}}$ is acyclic and there is no node v and no action f_{init} such that $v \rightsquigarrow f_{\text{succ}}$ and $v \rightsquigarrow f_{\text{fail}}$ are paths in $G_{C_{\mathcal{A}}}$.*

Algorithm 1: Viability, Fairness, and Entailment Test

input : a complex action C with assertions H and domain knowledge D

if $C \setminus (\mathcal{V}_a \times \mathbb{Q}^+ \times \mathcal{V}) \neq \emptyset$ **then** `/* ensure viability */`
  fail C cannot be reliably executed;

$(V, E, w) \leftarrow (\emptyset, \emptyset, \emptyset)$; `/* initialize graph structure */`

foreach $(v + d \leq v') \in C_{\mathcal{A}}$ **do** `/* populate graph */`
  $V \leftarrow V \cup \{v,\ v'\}$;
  $E \leftarrow E \cup \{(v', v)\}$;
  $w \leftarrow w \cup \{(v', v) \mapsto -d\}$;

if (V, E, w) *is cyclic* **then** `/* ensure fairness */`
  fail C is not fair;

if $v \rightsquigarrow f_{\text{succ}}$ *and* $v \rightsquigarrow f_{\text{fail}}$ *are paths in* (V, E, w) **then**
  fail C is not fair; `/* ensure entailment */`

if $C_{\mathcal{A}} \wedge C_{\mathcal{D}} \wedge D \wedge \neg H$ *is inconsistent* **then**
  fail the assertions H are not necessarily satisfied;

PROOF (SKETCHED). Paths in the graph correspond to temporal dependencies and because of $(\star)$ actions either succeed or fail. Accordingly, if there are cyclic dependencies in the graph or the execution of an action depends on the success and failure of one of its predecessors, the preconditions of the actions cannot be met and thus it is never executed and the complex action is not fair. □

The following theorem establishes a relationship between runtime traces which formalize the execution of actions and solutions of disjunctive temporal problems which are derived from the temporal dependencies of the complex action.

THEOREM 2. *A trace τ of a fair complex action C is a runtime trace iff the variable free formula $\tau(C_{\mathcal{A}} \wedge C_{\mathcal{D}})$ holds.*

PROOF (SKETCHED). By definition of T, the values in $\mathbf{T}$ satisfy the constraints from $C_{\mathcal{A}}$. Moreover, the definition of runtime traces excludes solutions of $C_{\mathcal{A}}$ that do not satisfy $(\star)$ which is formalized by $C_{\mathcal{D}}$. Accordingly, runtime traces satisfy $C_{\mathcal{A}} \wedge C_{\mathcal{D}}$ and conversely from every solution of $C_{\mathcal{A}} \wedge C_{\mathcal{D}}$ an appropriate scenario can be constructed that specifies a runtime trace. □

Definition 20. The assertions H of a complex action C with domain knowledge D are *entailed* iff for all runtime traces τ of C that conform to D the variable free formula $\tau(H)$ holds.

Accordingly, assertions are entailed, if they hold for every runtime trace of the corresponding actions, that is, if they hold for every way an action can be executed during runtime.

THEOREM 3. *The assertions H of a fair complex action C with the domain knowledge D are entailed iff the disjunctive temporal problem $C_{\mathcal{A}} \wedge C_{\mathcal{D}} \wedge D \wedge \neg H$ is inconsistent.*

PROOF (SKETCHED). Runtime traces correspond to the solutions of $C_{\mathcal{A}} \wedge C_{\mathcal{D}}$. However, not all runtime traces obey the restrictions specified by the domain knowledge. Accordingly, runtime traces that conform to the domain knowledge must satisfy D and thus they are solutions of $C_{\mathcal{A}} \wedge C_{\mathcal{D}} \wedge D$.

To verify that all those traces imply the assertions H, it suffices to show that $(C_{\mathcal{A}} \wedge C_{\mathcal{D}} \wedge D) \Rightarrow H$ is valid and thus that $C_{\mathcal{A}} \wedge C_{\mathcal{D}} \wedge D \wedge \neg H$ is inconsistent. □

These established connections are exploited by Algorithm 1 to check the fairness of actions and entailment of assertions by means of basic graph properties and the inconsistency of disjunctive temporal problems which can be verified in finite time by approaches like [5]. The comlexity of Algorithm 1 is dominated by the complexity of checking the consistency of DTPs which is known to be NP-hard [29].

6. COMPLEX ACTIONS IN CEP SYSTEMS

To obtain a generic approach that is applicable to a wide range of event processing systems, we represent the execution of actions by means of events and translate complex actions and reactive rules to regular complex event queries. In this way, we obtain the functionality of complex actions by actually evaluating a set of conventional event queries.

For the sake of simplicity, we assume that complex actions have a proper name and anonymous complex actions have been eliminated in a preprocessing step by introducing complex action rules for them.

6.1 Modelling Actions by Means of Events

Each action that is conceptually available in the language is mapped to four special types of events representing the request, initiation, success, and failure of the action. For instance, the `open-fire-dampers` action is represented by the following four types of events.

```
open-fire-dampers$request    open-fire-dampers$initiated
open-fire-dampers$succeeded    open-fire-dampers$failed
```

The payload of these events contains the parameters of the action and additional internal information, such as, a unique identifier to discriminate different instances of the same action and, if applicable, a reference to the identifier of the composite action that caused its execution. Moreover, the time-points associated with actions correspond to the occurrence times of the respective events.

6.2 Translation of Reactive Rules

Conceptually, reactive rules trigger the execution of actions. They are thus translated to deductive rules that derive `request` events which are distributed to the according actuators where they trigger the actual execution of the action. The reactive rule from Sec. 3.2 is thus converted to

```
DETECT
  adapt-ventilation$request{ payload{var Area} }
ON
  event e: fire{ var Area }
END
```

Recall that the identifier and the time of events are determined by the system. Accordingly, the identifier of the request event designates the identifier of the respective action.

6.3 Translation of Complex Actions

In the following, the basic ideas of translating complex actions will be introduced based on the complex actions `adapt-ventilation` from Sec. 3.2.

Temporal Dependencies. Temporal dependencies referring to certain time-points of actions occur for instance in the `where` part of actions and of event queries contained in the `succeeds on` part. Therefore, the contained references to actions need to be converted to references to events.

To this end, queries of events related to the corresponding actions are introduced and the references to time-points of actions are subsequently converted into references to the time of the introduced events. For instance, the query

```
event e: smoke{ area{var Area}, amount{var C} }
  where { C < 0.2, end(e)-init(a) < 30 sec }
```

taken from the `succeeds on` part of the `adapt-ventilation` action contains `init(a)` which refers to the initiation of the `open-fire-dampers` action. The query is thus converted to

```
and{
  event bot$init: adapt-ventilation$initiated{ id{var I} }
  event a$init: open-fire-dampers$initiated{ ref{var I} }
  event e: smoke{ area{var Area}, amount{var C} }
} where { C < 0.2, end(e)-end(a$init) < 30 sec }
```

Note that the join between the identifier of the complex action `adapt-ventilation` and the identifier referred by the initiation event of the `open-fire-dampers` action is mandatory to distinguish `open-fire-dampers` actions related to this instance of the complex action from those that just happen to be initiated at the same time and relate to other action instances or other complex actions.

In a similar manner, the preconditions $\{\perp_{\text{init}} \dot{+} 0,\ a_{\text{succ}} \dot{+} 0\}$ of the `activate-ventilators` actions from the same complex action are translated to the following query.

```
and{
  event bot$init: adapt-ventilation$initiated{ id{var I} },
  event a$succ: open-fire-dampers$succeeded{ ref{var I} }
} where { end(bot$init) + 0ms <= now(),
          end(a$succ) + 0ms <= now() }
```

It matches whenever the `activate-ventilators` can be initiated without violating any temporal dependencies of the complex action, that is, when all lower bounds that constrain the initiation of `activate-ventilators` are exceeded.

Action Composition. The semantics of composite actions specify that each sub-action is executed as soon as the lower bounds for their initiation are exceeded. To obtain the same behaviour by means of deductive rules, all sub-actions of a composite action are separated into independent rules, each of them responsible for the execution of one particular action in accordance with the temporal dependencies of the complex action.

To this end, the event query that monitors the preliminaries of each sub-action is determined as discussed above. These queries are subsequently included in the body of declarative rules that derive the appropriate `request` events. Note that the event query corresponding to the preliminaries of a sub-action always contains a query for the initiation of the complex action and thus the parameters required for the execution of sub-actions can be obtained from the payload of the queried initiated event.

```
DETECT
 activate-ventilators$request{ ref{var I}, payload{var A} }
ON
 and{
   event bot$init:
    adapt-ventilation$initiated{ id{var I}, payload{var A} }
   event a$succ: open-fire-dampers$succeeded{ ref{var I} }
 } where { end(bot$init) + 0ms <= now(),
           end(a$succ) + 0ms <= now() }
END
```

Accordingly, the execution of the `activate-ventilators` action is handled by the given deductive rule. Note that the query for `adapt-ventilation$initiated` events has been extended to extract the area that is passed to the action `activate-ventilators` as a parameter.

Execution Results. Translating the execution result specified in the `succeeds on` part of a complex action is straight forward. References to sub-actions in the `where` part of the query are rewritten as it has been described above and a deductive rule with the obtained query is created that derives the corresponding `adapt-ventilation$succeeded` event.

To obtain the query for the `failed` event, the same procedure is applied to the negation of the `succeeds on` query. To this end, the query in the `succeeds on` part needs to be timely bounded with respect to at least one time-point of an action. Due to space limitations, the simple but rather longish rules are only contained in the appendix of [17].

7. EXTENSIONS

Conditional Actions. Temporal dependencies of complex actions specify the execution order of actions relative to their initiation and success. However, in some situations it may be more suitable to specify the execution of actions in relation to the current state of the infrastructure, static data, or the occurrence of certain events instead of referring to the initiation and success of preceding actions.

This functionality can be added to Dura by means of conditional actions that block the execution of actions until a given event query matches. Recall, that in Dura event queries integrate queries for static and dynamic data.

```
compound{
  action a: request-operator-confirmation{ ... }
  IF event e: request-confirmed{ ref{id(action a)} }
       where { end(e)-succ(a) <= 20 sec }
  THEN action b: ...
}
```

The former action requests a confirmation from the operator and only executes the action `b` if the request is confirmed within 20 seconds. The translation of conditional actions corresponds to the translation of the execution result of actions. But instead of deriving `succeeded` and `failed` events, the corresponding `request` events are derived.

Disjunctive Temporal Dependencies. To obtain a deterministic specification of actions, the `where` part of complex actions must not contain disjunctive dependencies, such as

```
or{ succ(a) <= init(b), succ(a) <= init(c) }
```

However, if all temporal dependencies in a disjunction refer to the initiation of the same action, they correspond to a join of different execution branches and the execution of the complex action remains deterministic.

```
or{ succ(a) <= init(c), fail(b) <= init(c) }
```

The translation of temporal dependencies to event queries also applies for this kind of disjunctive constraints, it just needs to be slightly extended to ensure that the action is only initiated once, e.g., if in this example `a` is successful and `b` fails. The proposed semantic analysis can be reused by applying it to every disjunct of the disjunctive normal form of the corresponding dependencies. Accordingly, the analysis rejects the complete action if it rejects a single disjunct, as the dependencies must be met in any case.

8. RELATED WORK

Many approaches combining event detection and reaction rules have been proposed and studied by the research community. In fact, many of them are capable of specifying reactions to events in one way or another.

Complex Event Processing Most event processing systems support some kind of reactivity [19]. However, reactions are often dedicated to proprietary systems by means of remote procedure calls or some imperative host languages without a notion of complex actions. Notable exceptions are works from Paschke et. al. [23] and Behrends et. al. [3]. Although the proposed languages are capable of specifying composite workflows they still lack a language level support of complex actions that is tailored to physical actions and conveniently integrates complex events. Furthermore they do not provide a semantic analysis that verifies, e.g., fairness properties of actions.

Active Databases Active databases [1, 15, 14] realise automatic reactions in response to events by means of event-condition-action (ECA) rules. ECA rules are well established and have been intensively studied [24], but due to their origin in databases, events and actions are often related to (composite) updates of the internal knowledge base or trigger basic remote procedure calls. The authors of [8] argue that it is possible to realize workflows by means of ECA rules, but identify substantial shortcomings of hand coded rules that mimic imperative constructs like sequences.

Logic Based Formalisms The event calculus [27, 21] and the situation calculus [20] provide logic based frameworks which are commonly used for abductive planning and reasoning about the implications of actions based on a formal description of actions and their effects. Many extensions have been proposed that support composite actions and the specification of workflows [26, 11]. However, the focus of these formal and thus rather minimalistic formalisms is on reasoning about actions given a formal specification of their effects in contrast to high-level actions intended for emergency managers that are executed as events occur.

Temporal Constraint Satisfaction Temporal constraint satisfaction problems [12, 28] are commonly applied in problems related to planing and scheduling. In particular approaches that analyse the presence of dynamic plans that adhere to given constraints and cover events with contingent durations [31, 29] and disjunctions of events [30] are related to the semantic analysis of complex actions. However, our analysis focuses on the validation of fairness properties of actions that can fail during runtime, whereas those approaches try to determine dynamic plans for the execution of actions.

9. CONCLUSIONS

This work introduces complex actions with a clear separation of orthogonal dimensions and expressive temporal dependencies that naturally integrate complex events and actions in a high-level language. Thereby the specific particularities of physical actions as they are desirable, for instance, for emergency management have been considered. Moreover, we demonstrated how to realize complex actions by means of common complex event queries.

To compensate for inconsistent specifications of actions that are possible due to the desirable flexibility of our approach we have elaborated a semantic analysis that is tailored to physical actions and that scales with the in practice often varying degree of available knowledge on actions.

The proposed approach has been applied to implement the use cases of an emergency management related project [18]. And although this has been our main motivation for this work, our findings generalize to other applications that rely on external or physical actions.

10. ACKNOWLEDGMENTS

We would like to thank N. Eisinger and S. Brodt for valuable suggestions and discussions on our work. This work has been partly founded by the European Commission within the project "EMILI— Emergency Management in Large Infrastructures" under grant agreement number 242438.

11. REFERENCES

[1] R. Adaikkalavan and S. Chakravarthy. SnoopIB: interval-based event specification and detection for active databases. *Data and Knowledge Engineering*, 59(1):139–165, 2006.

[2] K. R. Apt, H. A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In *Foundations of deductive databases and logic programming*, pages 89–148. Morgan Kaufmann, 1988.

[3] E. Behrends, O. Fritzen, W. May, and F. Schenk. Combining ECA Rules with Process Algebras for the Semantic Web. In *Proc. Int. Conf. Rules and Rule Markup Languages for the Semantic Web*, pages 29–38. IEEE, 2006.

[4] M. Bettelini, S. Rigert, and N. Seifert. Optimum Emergency Management Through Physical Simulation— Findings from the EMILI Research Project. In *Proc. World Tunnel Congress*, 2013.

[5] S. Brodt and F. Bry. Temporal Stream Algebra. Technical report, University of Munich, 2012. `http://www.pms.ifi.lmu.de/publications/`.

[6] F. Bry and M. Eckert. Rule-based composite event queries: the language XChangeEQ and its semantics. In *Proc. Int. Conf. Web Reasoning and Rule Systems*, pages 16–30. Springer, 2007.

[7] F. Bry and M. Eckert. Rules for Making Sense of Events: Design Issues for High-Level Event Query and Reasoning Languages. In *AI Meets Business Rules and Process Management, Proc. AAAI Spring Symposium*. AAAI, 2008.

[8] F. Bry, M. Eckert, P.-L. Pătrânjan, and I. Romanenko. Realizing Business Processes with ECA Rules: Benefits, Challenges, Limits. In *Proc. Int. Workshop on Principles and Practice of Semantic Web*, pages 48–62. Springer, 2006.

[9] F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. *Proc. Int. Conf. Logic Programming*, 2401:255–270, 2002.

[10] D. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[11] N. K. Cicekli and Y. Yildirim. Formalizing Workflows Using the Event Calculus. In *Proc. Int. Conf. Database and Expert Systems Applications*, pages 222–231. Springer, 2000.

[12] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[13] K. Fridolf. Fire evacuation in underground transportation systems: a review of accidents and empirical research. Technical report, Lund University, 2010.

[14] S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using Petri nets. In *Proc. Int. Workshop on Research Issues in Data Engineering*, pages 2–9. IEEE, 1994.

[15] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model and Implementation. In *Proc. Int. Conf. Very Large Data Bases*, pages 327–338. Morgan Kaufmann, 1992.

[16] S. Hausmann, S. Brodt, and F. Bry. Dura: Concepts and Examples. Technical report, University of Munich, 2011. `http://www.emili-project.eu/index.php?id=481`.

[17] S. Hausmann and F. Bry. Towards Complex Actions for Complex Event Processing (Extended Version with Appendix). Technical report, University of Munich, 2013. `http://www.pms.ifi.lmu.de/publications/`.

[18] R. Llopis, X. Fust, J. a. González, J. L. Marín, N. Seifert, M. Bettelini, S. Rigert, V. Janev, P. Kroner, and D. Siller. Evaluation of our Simulation and Training Environment SITE and of our Use Case Implementations. Technical report, 2012. `http://www.emili-project.eu/index.php?id=544`.

[19] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison Wesley, 2001.

[20] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[21] R. Miller and M. Shanahan. Some Alternative Formulations of the Event Calculus. In *Computational Logic: Logic Programming and Beyond*, pages 452–490. Springer, 2002.

[22] Fire Investigation Summary Düsseldorf. Technical report, National Fire Protection Association, 1998.

[23] A. Paschke, A. Kozlenkov, and H. Boley. A Homogeneous Reaction Rule Language for Complex Event Processing. *Proc. Int. Workshop Event Driven Architecture and Event Processing Systems*, 2007.

[24] N. W. Paton and O. Díaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[25] N. Seifert, M. Bettelini, and S. Rigert. Concrete Use Case Models (Main Report). Technical report, 2011. `http://www.emili-project.eu/index.php?id=481`.

[26] M. Shanahan. Event Calculus Planning Revisited. In *Proc. European Conf. Planning*, pages 390–402. Springer, 1997.

[27] M. Shanahan. The Event Calculus Explained. In *Artificial Intelligence Today*, volume 1600 of *LNCS*, pages 409–430. Springer, 1999.

[28] K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.

[29] I. Tsamardinos, T. Vidal, and M. E. Pollack. CTP: A New Constraint-Based Formalism for Conditional, Temporal Planning. *Constraints*, 8(4):365–388, 2003.

[30] K. B. Venable and N. Yorke-Smith. Disjunctive temporal planning with uncertainty. In *Proc. Int. Joint Conf. Artificial Intelligence*, pages 1721–1722. Morgan Kaufmann, 2005.

[31] T. Vidal and H. Fragier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental and Theoretical Artificial Intelligence*, 11(1):23–45, 1999.