LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHR- UND FORSCHUNGSEINHEIT FÜR
PROGRAMMIER- UND MODELLIERUNGSSPRACHEN

Desugaring Dura

Compiling a High-Level Event Processing Language

Maximilian Scherr

Diplomarbeit

Beginn der Arbeit:	20. Juni 2011
Abgabe der Arbeit:	15. Dezember 2011
Betreuer:	Prof. Dr. François Bry
	Steffen Hausmann

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 15. Dezember 2011 ....

Maximilian Scherr

Zusammenfassung

Neuartige Ansätze für semiautomatischen, reaktiven Katastrophenschutz können dabei helfen, den sicheren Betrieb moderner Kritischer Infrastrukturen erheblich zu verbessern. Diese Ansätze bedürfen der Erkennung von Ereignissen und (abstrakten) Situationen, Einbeziehung verschiedener Zustände und Ausführung komplexer Workflows. Die Sprache Dura erfüllt diese Bedürfnisse da sie eine höhere Programmiersprache ist, die in sich die Konzepte von Ereignissen, Zuständen und Aktionen vereint.

Die Auswertung von Dura Programmen erfordert deren Übersetzung zu einer Algebra auf niedrigerer Sprachebene namens TSA. Ein monolithischer Compiler erscheint hinsichtlich Wartbarkeit und Modularität nicht wünschenswert, weshalb die Compilierung aufgeteilt wird in die Übersetzung von Dura zu deren Kernsprache Dura_C und die Übersetzung von Dura_C zu TSA, welche parallel zu ersterer verfolgt werden kann und aufgrund Dura_C's Nähe zu TSA weniger komplex ist.

In dieser Arbeit stellen wir Entzuckerungstransformationen vor, welche Duras syntaktischen Zucker in Dura_C Konstrukte umwandeln. Obwohl im Rahmen dieser Arbeit nicht die komplette Entzuckerung von Dura behandelt wird, erreichen wir die Übersetzung von Regeln zur Spezifikation komplexer Aktionen, sowie von verschachtelten Ereigniskompositionen, existentiellen Anfragen und anderem syntaktischen Zucker. Des Weiteren präsentieren wir nützliche Erkenntnisse für die zukünftige Fertigstellung des Dura Compilers.

Abstract

Novel approaches to semi-automatic, reactive emergency management can help to substantially improve the operation of modern Critical Infrastructures (CIs) in terms of safety and security. These approaches require the detection of events and (abstract) situations, integration of different states, and the execution of complex workflows. The language Dura fulfills these needs as it is a high-level language that consolidates the concepts of events, states, and actions.

The evaluation of Dura programs requires their compilation to a low-level algebra called TSA. A monolithic compiler seems undesirable in terms of complexity and maintainability. Therefore, the compilation is split up into the translation from Dura to its core language Dura_C and the translation of Dura_C to TSA which can be pursued in parallel and is less complex due to Dura_C's closeness to TSA.

In this thesis we propose desugaring transformations that turn Dura's syntactic sugar into Dura_C constructs. While not all of Dura's desugaring is covered within the scope of this thesis, we accomplish the translation of complex action rules, nested event compositions, existential queries, and other syntactic sugar. Furthermore, we present valuable insight for the completion of the Dura compiler in the future.

Acknowledgements

Without the continuous support of my kind advisor, Steffen Hausmann, who offered valuable guidance throughout my work on this thesis, I might not have achieved its completion. For this I am deeply thankful.

I would also like to thank Prof. Dr. François Bry for inviting me to write this thesis at the Teaching and Research Unit for Programming and Modelling Languages where I always felt welcome. After all, it was Prof. Bry's enthusiasm for improving emergency management that sparked my interest in the EMILI project. His input has always been kind and helpful, not only during this thesis but also in helping me decide on my future goals.

Of course, my gratitude extends towards all members of the Teaching and Research Unit for Programming and Modelling Languages. Among these I would particularly like to thank Dr. Norbert Eisinger who since early on during my studies at the Ludwig-Maximilians-University has repeatedly reassured me in my pursuits, both in classes as well as during consultations.

Thanks also to my family and friends who throughout my studies and my work on this thesis have provided invaluable support and company I will cherish forever.

Maximilian Scherr

Contents

1. Introduction **1**

2. Preliminaries **5**
2.1. The Language Dura . . . 5
2.1.1. Events . . . 5
2.1.2. Stateful Objects . . . 10
2.1.3. Actions . . . 10
2.2. Dura's Core Language: Dura$_{C}$ . . . 14

3. Translating Dura to Dura$_{C}$: Design and Basic Concepts **17**
3.1. Syntactic Sugar in Dura . . . 17
3.2. Translation Workflow . . . 18
3.2.1. Incremental Compilation . . . 18
3.2.2. Stepwise Desugaring of Dura . . . 20
3.3. General Translation Techniques . . . 20
3.4. Translating Complex Actions . . . 22
3.4.1. Inherent Difficulties with External Actions . . . 22
3.4.2. Constraining Action Executions . . . 23
3.4.3. Semantic Analysis of Executability . . . 25
3.4.4. Translation of Complex Action Rules . . . 28
3.5. Related Work . . . 30

4. Implementation and Algorithms **33**
4.1. Proof of Concept . . . 33
4.2. Semantic Analysis and Typing . . . 33
4.3. Transformation Sequence Overview . . . 35
4.4. Resolving Nested Action Compositions . . . 37
4.4.1. Nested Action Compositions . . . 37
4.4.2. Action Composition in Reactive Rules . . . 43
4.4.3. Complete Transformation . . . 44
4.5. Translating Complex Action Rules . . . 45
4.5.1. Complex Action Rules . . . 45
4.5.2. Handling Execution Constraints . . . 46
4.5.3. Translation to Reactive Rules . . . 50
4.5.4. Composition Translation . . . 59
4.5.5. Success and Failure Specifications . . . 59
4.5.6. Complete Transformation . . . 68

4.6. Simplifying Reactive Rules 69
4.6.1. Flat Execution Calls Reactive Rule 69
4.6.2. Simplification 69
4.6.3. Complete Transformation 71
4.7. Resolving Nested Event Compositions 72
4.7.1. Nested Event Composition in Positive Queries 72
4.7.2. Nested Event Composition in Negated Queries 86
4.7.3. Complete Transformation 89
4.8. Translating Existential Quantification 90
4.8.1. Existential Queries 90
4.8.2. Translation Idea 90
4.8.3. Translation 92
4.8.4. Complete Transformation 94
4.9. Replacing Aggregation Operations 95
4.9.1. Deductive Rule Pattern and Grouping in Head 95
4.9.2. Handling Aggregations in Body Supplement 96
4.9.3. Handling Aggregations in Head Term 97
4.9.4. Complete Transformation 97
4.10. Resolving Expressions in Rule Heads 98
4.10.1. Deductive Rule Pattern 98
4.10.2. Resolution 98
4.10.3. Complete Transformation 99

5. Future Work **101**
5.1. Limitations of the Implementation 101
5.2. Possible Extensions and Optimizations 102
5.3. Shortcomings of the Desugaring Approach 104

6. Conclusion **105**

A. Complex Action Rule Example **107**

B. Nested Event Composition Example **113**

1. Introduction

While in an ideal world it might be possible to prevent the occurrence of situations leading to emergencies entirely, this cannot be said for ours. The causes for natural hazards as well as man-made disasters such as terrorism, war, crime, and technological hazards, are manifold. Our inability to prevent all possible disasters at their root does not preclude us from attempting to prevent emerging emergencies. Quite to the contrary, the acceptance of the former can be a source of encouragement for the latter.

Although we might not be able to preempt disasters entirely, may they be big or small, we can strive for a limitation of their extent. Avoiding unnecessary risk for disasters, preparation (e.g. stockpiling emergency supplies), response (e.g. evacuation) and recovery (e.g. rebuilding infrastructure) can vastly mitigate a disaster's consequences. These aspects are to be found at the core of the discipline of emergency management [12].

The EMILI ("Emergency Managamenet in Large Infrastructures") project aims to find novel and improved ways for emergency management. This project is particularly concerned with large Critical Infrastructures such as power grids, airports, and metro systems. To this end, a new generation of control systems for these infrastructures, employing "the next generation of Web technologies like event processing, active Web, and Semantic Web", is to be envisioned [1].

The objective of these control systems is to support emergency operators during emergencies and exceptional situations in general. First and foremost, this entails the detection of such emerging situations. Subsequent swift reaction to these detected situations is crucial. However, overly rash decisions can hinder the emergency management efforts. It is thus the goal of these new control systems to back emergency operators in their decisionmaking, for instance by offering simulations on possible outcomes or enhanced visual presentation of data on sensors and the emergency response progress in a graphical user interface. Furthermore, emergency operators are vital, expensive personnel that should be enabled to focus mainly on important tasks, while an emergency management system is entrusted with performing simple, tedious routine tasks. To this end, such a system ought to offer facilities for semi-automatic reactions to given situations. These reactions incorporate the outcome of fast computable simulations, and may be composed of complex workflows requiring specification and manual (i.e. after a human decision was made) or automatic execution.

The metro scenario, one of the use cases of the EMILI project described in [25], outlines how various aspects of reactive emergency management are brought together for the treatment of a potential fire outbreak in an en-route train. On a train, located in the tunnel heading towards a station, a fire is detected and general immediate actions are initiated. These actions include, for instance, alerting medical and fire services, diversion of approaching trains, increasing the fresh air supply, and preparation of extinguishing water supply. Shortly after, the fire is categorized as being small due to temperature measurements be-

ing below a threshold value. A simulation is triggered which enables the prediction of the situation within the next couple of minutes. This in turn leads to appropriate, concrete response in the form of specific actions such as the organization of evacuation for the next platform the train will arrive at. Subsequently, the determined escape routes are announced to the passengers via the public announcement system of the arrival station, according to the smoke propagation estimated by the prior simulations. Other strategic actions, that are adapted to the predicted situation, include adjusting the ventilation system as well as enabling doors and lights on escape paths. The train stops at the next station and an increase in temperature is detected changing the categorization of the fire to "significant fire". This leads to a reconsideration of the emergency strategy and causes the preparation for evacuation of the adjoining metro stations.

To model the aforementioned scenario, sensor data needs to be observed in the form of *events* and needs to be correlated in a timely fashion. The *state* of the various parts of the infrastructure needs to be tracked and carefully observed. Furthermore, immediate *actions* need to be automatically executed in case of an emergency, which includes the initiation of simulations for determining the best evacuation strategy. One can see how the combination of events, states, and actions is crucial for a language designated to be used for describing the management of use cases such as the metro scenario.

One contribution of EMILI is the elaboration of a "high level, declarative and uniform reactive event query language"[5] called Dura, which in its core is a language for Complex Event Processing (CEP), i.e. the processing of events "in a continuous and timely fashion" [6]. The entailed concept of deriving "higher-level knowledge from lower-level events" lends itself to serving the formalization of knowledge required for emergency situation assessment and response. Dura is carefully designed to meet the requirements of modern emergency management as it is envisioned in the use cases [24] of the EMILI project. This means that in contrast to other event processing languages, which either support events *or* states, Dura integrates events, states, *and* actions [4].

Dura is a highly expressive language the programs of which need to be executed on top of a runtime system. However, this runtime system can only execute (or evaluate) a lower-level algebra, called Temporal Stream Algebra (TSA), which is suited for temporal streams [5]. Hence, programs written in Dura need to be translated to TSA prior to execution. However, Dura is not directly compiled into TSA, instead compilation has been split into sequenced stages. This incremental compilation approach has been chosen for two reasons, firstly to facilitate working on a prototype implementation in parallel and to improve maintainability, and secondly for ease of development on the compiler as a whole. Creation of an arguably monolithic compiler of the language Dura in its entirety is avoided as follows. A sub-language of Dura, called Dura_C (Dura Core), has been identified, which only contains the most basic constructs needed for reactive event processing. This makes it a lot closer to the lower-level expressions of TSA than Dura is, rendering the translation from Dura_C to TSA considerably easier [5].

Compared to Dura_C, Dura contains more syntactic constructs which allow for concise expression of complex concepts and workflows. During execution, these concepts and workflows would be broken down into simpler concepts, rules, or steps. According to [5] Dura_C "contains all mandatory constructs to remain as expressive as Dura whereas syntactic sugar is

omitted". Building on this notion, the topic of this thesis is to design and implement the desugaring of Dura, which is to remove all this syntactic sugar from a given Dura program, i.e. the translation of *correct* Dura programs to Dura$_C$ programs. The work on a Dura$_C$ compiler could commence in parallel to this thesis.

For further modularity we split up the desugaring process into a series of desugaring transformations, each of which is to remove *some* syntactic sugar. All of this is done on an abstract syntax tree which is the input to the desugarer, with the desugarer's output being again an abstract syntax tree that can serve as input to a Dura$_C$ compiler.

Among the most notable features of Dura not to be found in Dura$_C$ is the specification of complex actions. Much like complex events serve to derive high-level knowledge from simple events, complex actions can be used for higher-level abstraction on the (constrained) execution of simpler actions [4]. For instance, a complex "fire response" action might entail the ringing of a fire alarm, blocking access to (empty) elevators, evacuating, and *only after* evacuation was successful, shutting locks.

Regarding complex action specifications, it is particularly desirable to be able to detect early on, i.e. at compile time, whether all sub-actions that are part of a specified complex action will actually ever be executed. If, for instance, we know at compile time that the programmer has not specified when success of the evacuation action can be derived, we already know that the action of shutting locks in the aforementioned "fire response" action example will never be executed. For the type of complex action specifications covered in this thesis, we propose a semantic analysis that can detect such cases.

Another important aspect is the maintenance of action instance information, so that actions as part of a complex action can be reliably related to each other during execution. After all, the only tools available for this task in Dura$_C$ are event queries, simple reactive rules for concurrent execution of atomic actions, and certain premises on the behavior of the runtime system (i.e. events entailed by actions).

While a large part of this thesis has been devoted to the translation of complex actions, concentrating on issues pertaining to the constraining and synchronization of actions, it does not fall short of covering the translation of other helpful features of Dura such as nesting of composite queries and actions, arithmetic expressions and aggregation operations in places where Dura$_C$ forbids these, and not to forget, existential queries.

The organization of this thesis is described in the following. After this introduction, in chapter 2, the languages Dura and Dura$_C$ will be briefly introduced. This chapter cannot discuss them in their entirety due to space constraints and the reader is advised to consult earlier research reports on Dura [13] and Dura$_C$ [5] for a more extensive coverage. The focus of chapter 2 will lie especially on those language details that have slightly changed since Dura's first informal introduction in [13] and those language concepts that are most relevant for further discussion in the following chapters.

Chapter 3 will give an overview over the basic ideas that went into designing the translation from Dura to Dura$_C$, expanding on some of the aforementioned concepts. Subsequently, chapter 4 will cover the implementation of the desugarer as well as the translation patterns and algorithms used therein in detail.

A discussion on the limitations of the current implementation, possible optimizations, as well as an outlook on future work in general will follow in chapter 5. In the conclusion, forming the last chapter, we will reflect again on the achievements and shortcomings of the desugaring transformation approaches described in this thesis.

2. Preliminaries

Both the languages Dura and Dura$_C$ have been introduced in detail in Hausmann et al. [13] and Brodt et al. [5]. For an in-depth understanding of these languages and in extension this thesis, it is advised to consult these documents. Nonetheless, we find it important to introduce the basic language concepts and features for readers who are yet unfamiliar with Dura and Dura$_C$. This chapter mainly serves this purpose. It mainly focuses on syntactic constructs that have been specified more narrowly since the first introduction of Dura and those constructs that are important for a better understanding of this thesis. In this regard this chapter can be considered an extension of Hausmann et al. [13] as well as a summary thereof.

2.1. The Language Dura

Dura is a declarative, high level event processing language used for modeling complex events, states and state changes, reactions in the form of complex actions, and filtering of knowledge derived from events. This chapter briefly introduces the most important concepts and related syntactic constructs as well as some of their limitations.

2.1.1. Events

As can be expected from an event processing language, events lie at the core of Dura. Events are volatile messages about (but not limited to) change and carry specific data (e.g. temperature, location, water level, etc.) as well as general, implicit data such as their reception (or occurrence) time.

In Dura this event message data, called payload, is stored as semi-structured data terms of a restricted kind. These terms have a fixed schema that does not allow recursive definitions or subterm siblings with duplicate labels. Listing 2.1.1 shows an example payload omitting implicit data (e.g. ID, reception time, etc.).

Basic or simple events can be categorized into external or internal events, whereas the latter refers to those events generated (or derived) within the event processing system and the former refers to events which have their origin outside the event processing system itself, such as sensor data or user input. From these simple events so-called complex events can be derived by querying the event stream for matches. This is an important tool for abstraction and unifying representations of other events.

Listing 2.1.1: Payload example

```
temp{
  ⋮
  measurement{
    temperature{ 34 },
    unit{ "centigrade" }
  },
  area{ "cabin" }
}
```

2.1.1.1. Event Definition

For the event processing system and compiler to be able to efficiently handle events they need to be defined and given schemata. Schemata are described in more detail in [5] but to summarize, instead of providing values at leaves of a data term a schema provides type information instead. The basic type of event definitions has the form `EVENT` *schema* `END`. Listing 2.1.2 shows an example event definition of the `temp` event.

Listing 2.1.2: Event definition example

```
EVENT
  temp{
    measurement{
      temperature{ int },
      unit{ string }
    },
    area{ string }
  }
END
```

It is also possible to define custom (basic or composite) types. Again, this is not covered here but in [5].

2.1.1.2. Deductive Rules

For deriving complex events one needs to be able to specify what queries need to be matched, what data to extract and how to combine it. Deductive rules serve this purpose and are of the form `DETECT` *event* `ON` *query* `END`. They are found as extension to event definitions of the form `EVENT` *schema* `WITH` $rule_1$ ... $rule_n$ `END` specifying what rules lead to deriving the respective events. Listing 2.1.3 shows such a definition for the `pmet` event which states that for every `temp` event a `pmet` event containing the same data is derived.

Query terms do not need to be complete, i.e. not all data has to match. Recursive queries are not allowed. Furthermore, the head of a deductive rule may not specify implicit event data such as ID or reception time since this is created by the runtime system. The reception time of a derived event depends on the reception times of the queried events and is not the same as the actual (system) time at which the event was derived by the runtime system.

Listing 2.1.3: Deductive rule example

```
EVENT
  pmet{
    measurement{
      temperature{ int },
      unit{ string }
    },
    area{ string }
  }
WITH
  DETECT
    pmet{
      measurement{
        temperature{ var A },
        unit{ var B }
      },
      area{ var C }
    }
  ON
    event: temp{
      measurement{
        temperature{ var A },
        unit{ var B }
      },
      area{ var C }
    }
  END
END
```

Deductive rules need to be range restricted, a detailed explanation of which can be found in [5]. Informally, "range restricted" means that variables used in the head of a deductive rule need to be variables occurring in queries in the rule's body that are not hidden. Variables with same name need to match the same values. Queries my contain a supplement which defines new variables (`let`) or limits a query's matches (`where`). Furthermore, they can also be negated or existentially quantified.

2.1.1.3. Event Composition

In listing 2.1.3 the body of the deductive rule only contains one query. It is also possible to combine several queries by using conjunction (`and`) or disjunction (`or`). Such event composition can occur in the body of a deductive rule. Note that a lone event query in a rule's body can be considered an event composition with only one query. An event composition may also have a supplement which may contain multiple elements such as `where` and `let` blocks as well as groupings, i.e. `group by` blocks (only in the case of a conjunction).

Event composition can be nested but these nested compositions are somewhat restricted. As a general rule of thumb, if one uses the same variable inside and outside the nesting it must be exposed (or not hidden) at the end of the nested composition. In terms of range

restriction this can be rephrased as follows. If one were to look at the nested composition as the body of a deductive rule, not necessarily all variables in the body can be used in that rule's head. The variables outside a nested composition must be from this (imagined) set of variables exposed by the nested composition.

Listings 2.1.4 and 2.1.5 show two examples of forbidden nested compositions. The former contains a nested disjunction that cannot guarantee that any match of the combined query actually has a value for `var A` which needs to match with the outer variable `var A`.

Listing 2.1.4: Forbidden nesting (`or` case)

```
and {
  event: t{ p{ var A } },
  event: or {
    event: x{ p{ var A } },
    event: y{ p{ var B } }
  }
}
```

In listing 2.1.5 the same that is the case here for `not` would be the case for `exists`. Both negated queries as well as existential queries are considered negative. Note that the `where` block contains time constraints which are not specified here for the sake of brevity.

Listing 2.1.5: Forbidden nesting (`and` with negative subquery case)

```
and {
  event: t{ p{ var A } },
  event: and {
    not {
      event i: x{ p{ var A } }
    } where { ... },
    event j: y{ p{ var B } }
  }
}
```

Listing 2.1.6: Forbidden nesting (`group by` case)

```
and {
  event: t{ p{ var A } },
  event: and {
    event i: x{ p{ var A } },
    event j: y{ p{ var B } }
  } where { ... }
    group by { var B }
}
```

There is yet another case, shown in listing 2.1.6, that is not allowed, which can occur when grouping is involved. It is important to mention this specifically, as such a case appears as an allowed example in [5]. It is important to note that in fact we consider it *not* allowed due to the fact that the grouping `group by { var B }` causes the variable `var A` not to be

exposed to the outer composition and in turn to the query there that needs to match with this variable `var A`.

2.1.1.4. Event Accumulation

Event accumulation includes aggregation, negation, and existential quantification and requires the limitation of the time frames to be considered. Due to querying a possibly infinite stream of events it is impossible to query whether an event does not occur at all. After all, it could occur at some point in the future. Hence, such query could never be answered. The same is true for existential queries. However, it is possible to consider a negative query in relation to a positive query. Listing 2.1.7 shows an example of this.

Listing 2.1.7: Negative query example

```
and {
  exists {
    event i: x{ p{ var B } }
  } where {
    event i during event j,
    var B < var A
  },
  event j: y{ p{ var A } }
}
```

Note that for existential queries and negated queries the time frame limitation needs to occur in the negation's or existential quantification's supplement, i.e. `where` block. In fact this is the case for all expressions relating identifiers and variables inside of a negative query with positive identifiers and variables outside. This supplement is the only place the contained query can be related to the outside besides variable matches within the contained query itself. Moving the condition `var B < var A` to a `where` block of the composition is currently not allowed.

Listing 2.1.8: Aggregation example

```
DETECT
  q{ p{ sum(var B) * 2 } } group by { var A }
ON
  and {
    event i: x{ p{ var A } },
    event j: y{
      p{ var B },
      q{ var C }
    } let { var int D = var C + 4 }
  } where { event j during event j }
    group by { var A, var B }
    where { avg(var D) > 10 }
END
```

For grouping and aggregation it is merely necessary that a time frame limitation (in a `where` block) happens before the grouping in a `group by` block, both occurring in the same supplement. After a `group by` block one may aggregate with an `aggregate` block or by using aggregation operations in `where` or `let` blocks.

In Dura it is also allowed to group in the head of a deductive rule and use aggregation operations in the leaves of the head term. These leaves may also contain arithmetic expressions. In the body of a rule arithmetic expressions are only allowed in conditional statements (`where` blocks) and variable definitions (`let` blocks). Listing 2.1.8 shows an example using the features described above.

2.1.2. Stateful Objects

Modeling state is very useful for reactive emergency management. For instance, depending on the control system's state, events might need to be treated differently. State is modeled in Dura by so-called stateful objects which unlike events are non-volatile in nature. However, similar to events they carry specific data and implicit data including the so-called valid time. Valid times are time intervals to each of which there exists a data term for the associated stateful object. Stateful objects can be modified and every time a stateful object changes a new interval with (at the time) unknown end time is created and associated with the new properties. The data used in stateful objects is the same kind of semi-structured data used for events and each stateful object has a fixed schema. Hence, stateful objects are defined just like events with stateful object definitions of the form `STATEFUL OBJECT` *schema* `END`.

Changes to stateful objects are announced by the runtime system in the form of entailed events that are not described in detail here. More importantly, stateful objects can be queried themselves like events. Instead of event queries (`event:` ...) one uses state queries (`state:` ...) which behave mostly like the former. One important difference between events and stateful objects is that while the former can be detected only after its ending is equal or before the current runtime system time, stateful objects can be queried and "detected as soon as they have been created" [13].

One of the uses of stateful objects within Dura presented in [13] is the structuring and modularization of rules. This can be done with the `WHILE` *stateQuery* `LET` $rule_1$`,` ...`,` $rule_n$ `END` statement. The contained rules only cause derivation of events if the queried state is valid at the end of the events queried in the rules. This feature is not covered in this thesis and might require further specification.

With help of the `DERIVE` *statefulObject* `FROM` *stateQuery* `END` statement it is possible to derive stateful objects from others, serving a similar function as the deductive rules do for events. Stateful object derivations can be considered "database-like views on stateful objects" [13]. This feature is also not covered in this thesis and thus not described in further detail here.

2.1.3. Actions

Actions are the basic means of response to events in Dura. They occur either as basic actions, i.e. non-decomposable, simple commands, or as complex (or composite) actions, i.e.

actions composed of other actions possibly further constraint on their execution order or other aspects. Actions can be further categorized into internal and external actions. The former refers to actions executed within the runtime environment of the event processing system, the latter to those executed outside of it, for instance by attached infrastructure systems. Hence, external actions cannot be executed directly. They can only be initiated by the event processing system. Subsequently, an external actuator has to perform their actual execution.

Similar to events and stateful objects, actions have a payload in the form of semi-structured data (with schema), which in this case contains the parameters for execution. This payload is preserved in the events entailed by action execution, namely *actionName*`$initiated{ ... }`, *actionName*`$failed{ ... }`, and *actionName*`$succeeded{ ... }`. The payload can be found in the `payload` subterm of these events. More on events entailed by actions can be found in [13] with a slightly different syntax. The change in syntax was necessary due to the introduction of schemata which would, for instance, not allow for a general `action-initiated{ ... }` event for all actions.

2.1.3.1. Action Definition

Just like events and stateful objects, actions need to be defined. This is done in Dura with the construct `ACTION` *schema* `END`. Listing 2.1.9 shows a basic definition for the external `evacuate` action.

Listing 2.1.9: Action definition example

```
ACTION
  evacuate{
    area{ string }
  }
END
```

In this definition form, an execution of this action would never be considered successful, i.e. the event `evacuate$succeeded{ ... }` would never be derived. Dura allows for specifying success within the action definition. Listing 2.1.10 shows an example for this. The `succeeds on` block contains a query that specifies when an execution of the action is to be considered successful. The earliest match and only this causes the execution to be considered successful.

The query `event i: action$initiated{ ... }` is special as it refers to exactly the instance of the action for which success shall be derived. Replacing it with the query `event i: evacuate$initiated{ ... }` would cause any execution of the `evacuate` action to be considered successful as soon as there was *some* evacuation confirmation for *some* other action `evacuate` instance.

These instance references will be revisited later but one important limitation for their usage is introduced here. Due to their later translation one needs to limit them in terms of range restriction. Consider instance references (even different ones) as if they were variables with the same name. If this "variable" is used somewhere in a possibly nested event composition

Listing 2.1.10: Action definition with success specified example

```
ACTION
  evacuate{
    area{ string }
  } succeeds on {
    and{
      event i: action$initiated{
        payload{ area{ var A } }
      },
      event j: evacuation-confirmation{ area{ var A } }
    } where { j after i }
  }
END
```

Listing 2.1.11: Illegal instance reference example*

```
ACTION
  evacuate{
    area{ string }
  } succeeds on {
    and{
      event i: or {
        event: action$initiated{}
        event: temp{}
      },
      event j: evacuation-confirmation{}
    } where { j after i }
  }
END
```

it needs to be exposed to and beyond the outer composition in the `succeeds on` block. Listing 2.1.11 shows a counterexample[1].

It is also possible to specify when an action execution is to be considered failed by using a `fails on` block. It has the same syntax and features of the `succeeds on` block with the sole difference being that the contained query specifies failure instead of success. It is currently possible to write queries in both the `succeeds on` block and the `fails on` block that could cause an executed action to be considered as both successful and failed. It is the programmers obligation to make sure that this does not happen by writing appropriate queries.

2.1.3.2. Action Composition

Actions can be composed into complex (or composite) actions by Dura's facility of action composition. Like nesting event composition, action composition can be nested as well. There are three types of composition, i.e. `and`, `or`, and `concurrent`. The former two are

[1]The asterisk (*) in the caption signals the fact that this code is not allowed

described in [13] but have not been implemented yet. Here we describe the most general action composition using `concurrent`.

In its basic form `concurrent` composition merely causes the contained actions to be initiated pseudo-concurrently. This can be extended by providing a `where` block in the composition's supplement which contains execution constraints. What type of constraints is allowed there is described in section 4.5.2. In general it is also possible to allow the temporal relations *actionA* `before` *actionB* and *actionA* `after` *actionB* but this feature has not been implemented yet.

In complex action rules and in nested action compositions the supplement of an action composition may also contain `succeeds on` and `fails on` blocks. As is the case for such blocks in action definitions, they may contain action instance references.

2.1.3.3. Complex Action Rules

Complex action rules allow for specifying user-defined actions and are of the form `FOR` *action* `DO` *actionComposition* `END` to be used as part of action definitions. Extended action definitions `ACTION` *schema* `WITH` *rule* `END` (without `succeeds on` or `fails on` blocks) may contain only one complex action rule.

Listing 2.1.12 shows an action definition with complex action rule for the user-defined `open-evacuation-path` action. Only after the lights on the given path have been turned on successfully shall the doors to the evacuation path be opened. Furthermore an execution of this whole action is to be considered successful only after the doors have been opened.

Listing 2.1.12: Complex action rule example

```
ACTION
  open-evacuation-path{ path-id{ int } }
WITH
  FOR
    action a: open-evacuation-path{ path-id{ var A } }
  DO
    concurrent {
      action x: turn-on-lights{ path-id{ var A } },
      action y: open-doors-to-path{ path-id{ var A } }
    } where {
      // action x before action y
      end(action x) - begin(action y) <= 0 ms
    } succeeds on {
      event: action y$succeeded{}
    }
  END
END
```

Action instance references can refer to actions with identifiers within the complex action rule, but have the same limitation as discussed earlier for their usage in action definitions.

Conditional statements (IF ... THEN ... ELSE ... END) within complex action rules (as well as reactive rules) and timed success specifications are also mentioned in [13] but are not considered in this thesis.

2.1.3.4. Reactive Rules

Reactive rules are the declarative interfaces between events, stateful objects and action executions. They are of the form ON *query* DO *action* END, where the body of reactive rules contains a query (possibly with event composition) and the head contains an action or action composition without `succeeds on` or `fails on` blocks[2].

Listing 2.1.13: Reactive rule example

```
ON
  and {
    event: smoke-detected{ area{ var A } },
    event: fire-detected{ area{ var A } }
  }
DO
  concurrent {
    action  : call-fire-services{ to{ var A } },
    action x: evacuate{ area{ var A } },
    action y: concurrent {
      action: stop-ventilation{ area{ var A } },
      action: shut-locks{ area{ var A } }
    }
  } where {
    end(action x) - begin(action y) <= 10 min
  }
END
```

When a specified complex event is detected (i.e. the rule's query matches) the actions specified in the head (or execution part) of a reactive rule are initiated as the action composition dictates. Since the time an event is detected is equal or after the reception (or occurrence) time of the event, during runtime the actions are not guaranteed to be initiated right at the end of the modeled event reception time. Listing 2.1.13 shows an extensive example of a reactive rule that showcases the described features including nested action composition.

2.2. Dura's Core Language: $Dura_C$

The core language of Dura is described in detail in [5]. However, having summarized Dura above it is fairly easy to summarize $Dura_C$ (pronounced "Dura Core") by listing all the features of Dura it lacks. It has been asserted that all limitations of $Dura_C$ "are only of a syntactical nature [and] Dura rules can always be rewritten to (semantically) equivalent $Dura_C$ rules"[5]. The main Dura features missing in $Dura_C$ are:

[2]Nested compositions may contain such blocks.

- *Nesting of compositions* of any kind is not possible in Dura_C. There may only be one layer of action or event compositions.
- *Existential queries* are not supported.
- *Heads of deductive rules* may not contain groupings, arithmetic expressions, or aggregation operations.
- *Aggregation* in Dura_C can only be performed in `aggregate` blocks.
- *Complex action rules* cannot be defined at all.
- *Action composition* is possible but only in its simplest form, i.e. execution of the contained actions cannot be constrained and success or failure cannot be specified.
- *Action definitions* are allowed only in their simplest form, i.e. without success or failure specifications.

Furthermore, Dura_C lacks stateful object derivation (`DERIVE` ... `FROM` ... `END`), conditional statements (`IF` ... `THEN` ... `ELSE` ... `END`), and the `WHILE` ... `LET` ... `END` statement. The other limitations listed in [5] are currently under evaluation.

3. Translating Dura to Dura_C: Design and Basic Concepts

One important goal and feature of Dura is achieving a high level of abstraction. In particular this means that the language offers many elements that facilitate modeling complex events, states and actions. Many Dura constructs for this purpose can be identified as syntactic sugar that would make a direct compilation to the target language TSA (Temporal Stream Algebra) a difficult and monolithic task due to the need for considering many varieties of expression at once.

In order to counter this issue, Dura was stripped down to its core, namely the language Dura_C. Compilation of Dura can then be split into two translation tasks, i.e. translation from Dura to Dura_C and subsequent translation from Dura_C to TSA or possibly a different target language. As long as the respective interfaces are defined properly this approach is expected to result in modular, extendible, and maintainable code.

For the attempt of mapping the additional features offered by Dura to Dura_C we employ certain concepts and design decisions that are informally introduced in this chapter.

3.1. Syntactic Sugar in Dura

Some constructs in Dura immediately appear to be syntactic sugar for slightly restricted constructs in Dura_C. Take, for instance, grouping (`group by`) in deductive rule heads. The sample code in listing 3.1.1 is not valid in Dura_C as it only allows grouping in the body of deductive rules. However, simply moving the grouping (`group by { event i }`) to the end of the rule's body resolves this issue.

Listing 3.1.1: Grouping in head example

```
DETECT
  p{} group by { event i }
DO
  and {
    event i: t{},
    event j: u{}
  } where { event j during event i }
END
```

This reoccurring trait of having several equal ways of expression in Dura while Dura_C only offers very few choices comes as no surprise as Dura_C was designed to be expressive enough for creating event processing rules for the abstract machine, yet restricted in order

to facilitate translation to TSA [5]. As such is the premise, at the outset of this thesis we considered all additional features of Dura that are manifested as additional syntactic constructs, as syntactic sugar. Hence, translating Dura to Dura$_C$ means "desugaring" Dura.

Whether this approach is a sensible one merits some discussion. Firstly, consider the term "syntactic sugar". Often this term is used in reference to shorthand alternatives to basic language features the translation of which is usually trivial. For instance, the `+=` operator found in many imperative languages (e.g. C++, Java, Python) is shorthand for an addition operation and subsequent assignment. While some Dura constructs clearly fall into this category, others do not necessarily have such a trivial translation. Comparing with our `group by` example presented earlier, at the other end of the spectrum we find complex actions or action composition, the translation of which is not trivially obvious. However, the term "syntactic sugar" is not necessarily limited to referring to small, trivial additional constructs. In its most general usage the term "syntactic sugar" may very well refer to any "special syntactic forms that are simply convenient alternative surface structures for things that can be written in more uniform ways"[2].

A more intricate issue might be the fact that due to type-dependent elements of the languages Dura and Dura$_C$ such as schemata, typed assignments and custom types, some translations require a rudimentary knowledge derived from semantic analysis. This means that not all "syntactic sugar" constructs in Dura are of a purely syntactic nature. Since, at least for the construct translations described in this thesis, this additional semantic knowledge is not hard to derive we feel that the term "syntactic sugar" still applies to our situation.

Besides the discussion on proper terminology above, one has to consider the overall practicability of the desugaring approach. Although this approach is the premise of this thesis and we further pursued it, at times the approach can very well be challenged and the question of practicability will be revisited throughout the rest of this thesis.

3.2. Translation Workflow

The compilation of Dura is a large endeavor that has been split up into different phases. The scope of this thesis only covers a part of the compilation process, yet this part depends on other parts of the entire project and is itself a prerequisite for other parts. This section shall introduce the thoughts that went into the splitting of the compilation project and where the work of this thesis is placed within the entire compilation project.

3.2.1. Incremental Compilation

The main purpose of the language Dura is to offer an abstract and fairly easy way to create programs for complex event processing. As such it should contain an array of features that facilitate expressing the various rules and definitions for such programs. While it would very well be possible to create a compiler that directly translates Dura to TSA, it is, while not necessarily so, highly likely to come with a set of issues such as reduced maintainability

and modularity. Additionally, there would be no way to use any feature of the language whatsoever until such a direct compiler has been finished.

By first stripping Dura of its advanced features (which we identified as syntactic sugar in section 3.1) and implementing a compiler for Dura_C while in parallel creating a translation for Dura to Dura_C, one can avoid most of these issues, at least in principle. While Dura_C does not offer many features of Dura and can be considered less easy (or sweet) to use, it is expressive and abstract enough so as to lend itself to the creation of complex event processing programs. So, even before the entirety of a Dura compiler is implemented programmers can get familiar with its general concepts by using Dura_C. In fact this was the case for this project as the first prototype of the Dura_C compiler had been completed before the prototype implementation's work for this thesis was finished.

Figure 3.2.1 shows a diagram of the initial steps involved in this incremental compilation. The general workflow of the translation from Dura to TSA resembles a general compiler's pattern (cf. [3], p. 10). In the initial phase the Dura code is parsed and an abstract syntax tree (AST) representation is created (lexical and syntactic analysis). This phase is followed by a semantic analysis that in the case of Dura is supposed to find type and schema mismatches and generally check the soundness of Dura programs. Additionally it should save its findings such as variable-to-type mappings and provide this information to later phases, either by annotating the AST or additional data structures available to the following phases. At the time of writing this phase had not been implemented, so those few parts of semantic analysis information required during the translation from Dura to Dura_C had to be partly implemented. Section 4.2 contains more information on how this issue was worked around.

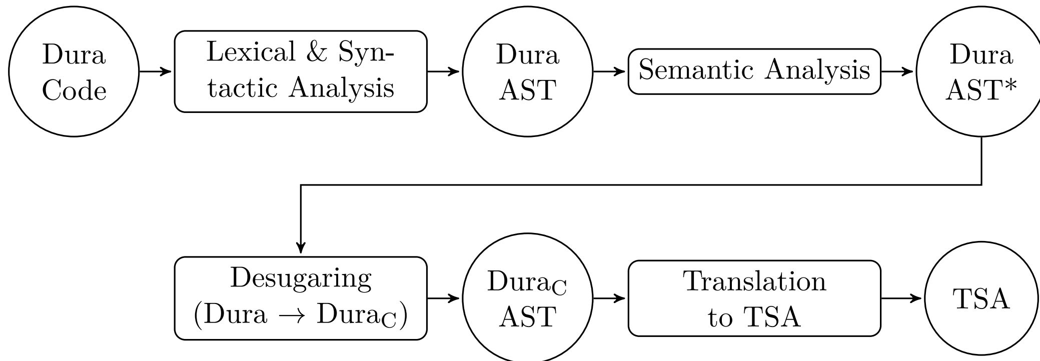


Figure 3.2.1.: Compilation steps

The semantic analysis phase is followed by the phase of translation from Dura to Dura_C, i.e. desugaring as explained in section 3.1, the topic of this thesis. The result of this desugaring process is a Dura_C AST. After desugaring, the original AST has been stripped of all constructs that are not found in Dura_C or rather, the Dura constructs have all been transformed into such constructs accepted by Dura_C. In a way, the desugaring algorithm as described in this thesis formalizes the semantics of Dura in terms of Dura_C.

The consideration of the translation workflow in this section ends with the translation of Dura_C to TSA by means of a developed Dura_C compiler, as mentioned earlier. Further

steps of compilation such as optimization and possibly further translation to other languages might follow this phase but are not considered in detail here.

3.2.2. Stepwise Desugaring of Dura

Ideally, the abstract syntax tree input to the desugaring process is already type and error checked. Currently this is not entirely the case and some portion of semantic analysis has been mixed in with the desugaring process. However, these workarounds can be relayed entirely to the earlier step of semantic analysis once it has been implemented. As per the initial plan and concept which is not negated by this temporary incompleteness of the compiler prototype project, for the general process of desugaring we can assume that this checking is indeed performed entirely before the desugaring process begins.

In its implementation desugaring itself is performed stepwise, i.e. it is realized by a sequence of transformations on the (checked) Dura abstract syntax tree. Each transformation brings the AST one step closer to the Dura_C AST. Hence, after every step we find an output AST representing a sub-language of the transformation's input AST, both being sub-languages of Dura.

While the order of those desugaring transformations is important they should be designed as independent as possible from each other. This refers mostly to the fact that except for the input AST, data structures resulting from semantic analysis, and minimal bookkeeping data structures, these transformations shall not share access to other data. This has the disadvantage of potentially walking the AST more than would be necessary in an optimal setting, e.g. collecting the same constructs several times. However, the overall AST changes in every transformation and so for sharing data resulting from already walking an earlier AST, bookkeeping and maintenance is required. The advantage of considering the input AST mostly isolated is an increase in modularity. For instance, when in the future Dura is extended with new features one only has to find out between which (desugaring) transformation steps a new transformation has to be added without having to take many data maintenance aspects into consideration.

Especially in the early phase of prototyping and testing with language features we find the possibility to rearrange and exchange some of the transformations beneficial. The initial focus is set on designing a functional prototype that is easy to maintain and extend. Optimization is still an option for a later version once the entire Dura language environment has matured and stabilized. Already during the course of this thesis and prototype implementation of those desugaring transformations this modular approach has proven to be useful. At some point slight changes to the AST representation of Dura constructs had to be introduced. Transformation involving those constructs could be swiftly adapted without the need to consider any of the other transformations.

3.3. General Translation Techniques

The Dura desugarer takes an AST and returns an AST that only contains constructs valid in Dura_C. Alternatively, the desugarer code could be designed to return Dura_C code that

would have to be parsed again for the compilation to continue. However, since the structures of a Dura AST and a Dura_C AST are extremely similar, transforming the input AST into a suitable output AST is in fact both easier and overall more efficient than generating Dura_C code.

In the course of this transformation, parts of a program's overall abstract syntax tree will change and close attention has to be payed in which order certain changes are applied to an already changed syntax tree.

Returning to our earlier "grouping in rule head"-example (cf. listing 3.1.1), transforming the syntax tree created from this code snippet and subsequently outputting it as code would result in listing 3.3.1, which is valid in Dura_C.

Listing 3.3.1: Grouping in head example transformed

```
DETECT
  p{}
DO
  and {
    event i: t{},
    event j: u{}
  } where    { event j during event i }
    group by { event i }
END
```

Consider the partial abstract syntax trees in figures 3.3.1 and 3.3.2 which represent the abstract syntax trees of a deductive rule that contains a grouping in the head and its desugared version in Dura_C respectively. The *grouping* node found in the head is removed from the children of the *head* node and appended to the children of the *body* node. Keep in mind that this is only a very simplified presentation for illustration purposes and the actual generic AST representation looks different.

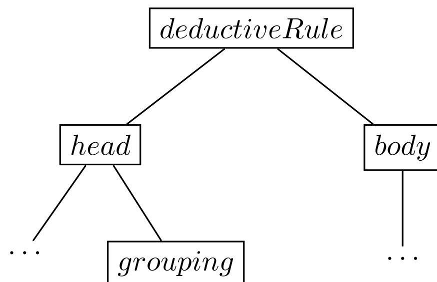


Figure 3.3.1.: Grouping in head (Dura)

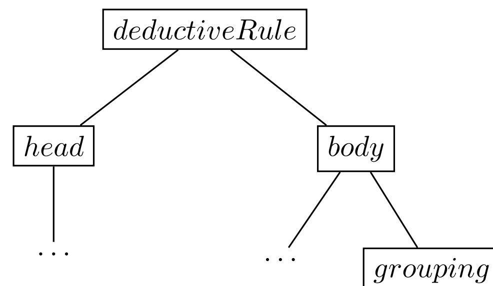


Figure 3.3.2.: Grouping in body (Dura_C)

Many desugaring transformations share similar concepts and perform similar subtasks. The following operations are among those used repeatedly as part of the desugaring process:

Moving & rearranging of constructs: Syntactic constructs are removed from their original place and moved to a different one. This refers for instance to what was performed in the aforementioned example about groupings.

Substitution of constructs: Constructs are replaced by different constructs. This is often combined with rearranging. For instance, an arithmetic expression `3 / (5 + (var A))` is replaced by a new variable `var X` which in another part of the syntax tree might be defined in a so-called `let`-statement as `var int X = 3 / (5 + (var A))`.

Extraction of (nested) structures: This is similar to substitution and refers to naming large structures and moving them outside of their original context. Usually this also involves a reference to the named construct where it originally occurred. One notable example for this is the resolution of nested event compositions.

Schema construction: Some changes require the construction of a new schema for a new event, stateful object or action and requires rudimentary semantic knowledge, e.g. the type of a certain variable within an extracted construct.

Creation of unique names: When giving new names for rules or changing identifiers one has to maintain that these new names do not clash with other names in their scope. For instance, when substituting the arithmetic expression `3 / (5 + (var A))` it would be unwise to substitute it with `var A` as it would lead to a potentially erroneous definition `var int A = 3 / (5 + (var A))`.

This rough list is non-exhaustive and merely presents common ideas used in the detailed algorithms described in chapter 4. In some cases constructs are even removed entirely and represented by different ones. One such example is the translation of complex action rules and their contained action composition.

3.4. Translating Complex Actions

For some of the syntactic sugar in Dura one can find similar, yet restricted, constructs in Dura_C but for other constructs in Dura one is hard pressed to find a suitable way to express them in Dura_C. One such construct of Dura is the non-trivial composition of atomic actions to complex actions.

Here, non-trivial refers to the fact, that while Dura_C allows for specifying reactive rules that cause the initiation of several actions (concurrent action execution), the language does not allow for specifying an execution order on those actions. Furthermore, Dura_C does not offer any facility at all for creating user-defined complex actions (via complex action rules).

Due to the difficulty in expressing such complex actions in Dura_C and the important role they play in the goals set for Dura [13], this section will introduce the considerations and ideas used for their translation, which is described in detail in chapter 4.

3.4.1. Inherent Difficulties with External Actions

Much of the difficulty in translating complex actions lies in the nature of external actions themselves. The only control a Dura programmer has over actions is their initiation. During runtime, initiation of an action is typically triggered by a firing reactive rule. This initiation is subsequently sent to a so-called actuator which starts the actual execution (cf. [5], p. 50). Moreover, if an action was initiated, this initiation is detectable and can for instance be used

in queries of deductive rules. For the success or failure of an action execution this guarantee cannot be made, due to the fact that actions are not required to give this feedback. Even if an action is specified to give this feedback one has to consider the case that, though specified, the conditions required for a successful (or failed) end might never be satisfied.

In general no estimates can be made on the duration of an action neither at compile time nor during runtime at the time of initiation. It could take between no and an infinite amount of time until the end of an action can be detected. Note that the actual end of an external action execution is not necessarily detectable by the event processing system. Take, for instance, an action with the only task of turning on an emergency light switch. While actually it is finished after it has sent the signal to the light switch this does not automatically mean success. In order to let the event processing system know when the action is successful one needs to declare its success by means of an event query which specifies when the action can be considered successful in the event processing system. Success might be more than a single positive responses to an execution. For instance, evacuation of an area is not successful just because we know that the emergency doors opened and people were instructed to evacuate, but when there is additional confirmation that no people remain in the area.

These properties restrict us in our possible solutions for realizing complex actions. For instance, if we did actually know the maximum duration of actions we could extend the range of allowed types of execution constraints to be imposed on a collection of composed actions. Knowing the duration would also allow us to use already available solutions to the so-called Simple Temporal Problem with Uncertainty (STPU) [17, 16, 15]. These solutions include checking for so-called controllability of plans, i.e. whether a collection of actions of uncertain but bounded duration can be executed according to given constraints and depending on observations during execution, and creating strategies for their execution. Since Dura's actions are not restricted to maximum durations and since additionally required real-time and latency guarantee properties (cf. latency accumulation in [18]) are currently not planned for in the abstract machine, i.e. the runtime environment of the event processing system, such solutions will not be employed.

Due to the nature of actions in Dura one is limited in reasoning about their execution. Yet, this limitation means that Dura_C and the current abstract machine do in fact provide sufficient tools for expressing action compositions, as described later in this thesis. However, it is important to keep in mind the fact that they might cease to do so if the properties of actions and the abstract machine were to be extended.

3.4.2. Constraining Action Executions

A crucial feature of action composition in Dura is the facility to exert control over the execution of actions encompassed by a composite action, by means of a conjunctive set of execution constraints. However, due to the properties (and inherent difficulties) of actions one is limited in the possibilities to express execution constraints in Dura as described in the following.

Since ultimately we only have control over the initiation of an action, only those constraints restricting such initiation in regard to an observed (past) initiation or proper end of other

actions are allowed. Currently we only consider those actions that ended successfully as properly ended, but the described concepts and ideas could be extended so that one can also restrict in regard to ends of failed action executions. For the time being we do not cover conditional constructs within an action composition, which would allow for further, dynamic control over when an action is to be initiated.

Another reason for restricting the allowed constraints is that they give rise to strictly defined semantics. It is not immediately obvious that this is an advantage and it could very well be considered a disadvantage. Take for instance the following example that is not allowed under the imposed restrictions:

- Let a and b be actions to be executed.
- Action b shall end successfully after action a has ended successfully.

Note that in this example we restrict the end of action b in regard to the end of action a. This constraint could very well be satisfied by ensuring that action b is only initiated after the successful end of action a has been detected. The question is whether it would be sensible to allow such constraints that indicate control over something that is only indirectly controllable. The danger of this would arise in the form of potential misunderstandings by Dura programmers and the automatic adjustments.

Let action a stand for "bring the fire department" and let action b stand for "open emergency doors". In most cases the fire department will arrive long after the emergency doors have been opened. So, although in this case it is not necessary, a programmer might see no harm in imposing the constraint that the fire department shall arrive after the emergency doors have been opened. However, due to that constraint and the automatic execution arrangement (i.e. always execute b after a), assuming that opening the emergency doors takes x minutes, the fire department will always arrive x minutes later than it could. This is a kind of overspecification on part of the programmer and is of course not to be blamed on the execution system. While a programmer might know or assume details about the tasks of an action, the compiler and in extension the execution system is mostly oblivious to those. That is why we believe it is sensible to only allow direct constraining of the initiations of actions.

Note that as opposed to constraining execution, checking of such constraints is indeed possible. A programmer might wish to specify that an execution of the complex action encompassing a and b is only to be considered successful if b ended after a. This can be specified in Dura specifically (i.e. using `succeeds on`) but is to be seen separate from constraining execution.

A further reason to restrict the constraint types as described is the facilitation of error debugging. Assuming that during a simulation run delay issues as the one described above occur, it is arguably harder to find their cause if one first has to prompt the system to output its automatically derived action execution order and adapt its base code than it would be to only adapt the code as is. If one is only allowed to constrain the initiation of an action (as is the case) one can immediately see how this initiation (directly) depends on other actions and their respective initiation or end.

In regard to sequencing action executions, a further restriction is imposed on constraints. Only enforcing lower bounds on the duration between a time point and the initiation of

an action is allowed. This means, that while it is fine to constrain by saying "initiation of action b shall wait *at least* five seconds after the end of action a", one is not allowed to constrain it saying "initiate action b some time after action a but initiation of action b shall wait *at most* five seconds after the end of action a". It might not be possible to satisfy this constraint as the initiation of actions might be delayed by the event processing system. Therefore such constraints cannot be guaranteed to be met. The property of having this guarantee is important for constraints used for planning the execution of actions as part of a complex action. For such guarantee to be made for the latter constraint, the execution mechanism within the runtime environment of the event processing system would have to be extended. But even then it might remain flawed unless real-time system properties can be guaranteed. This is a change that would have to be made several layers of abstraction and implementation below what is and can be considered within the scope of this thesis as it is only concerned with translating from Dura to Dura_C. Yet, it is an important point to be made and one potential disadvantage of the desugaring approach, at least given the current conditions.

If one desires to specify that execution of a complex action should be considered as failed if certain upper bounds on the duration between related time points are not satisfied during execution (at runtime) one can do so with specific constructs (i.e. using `fails on`). So, after the fact, e.g. the duration between two initiation events has been observed to be longer than it should have been, one can derive failure.

The issue of limiting the allowed constraints is not entirely discounted. It is quite possible that some of the mentioned limitations will be lifted after revisiting the issue in a future version of Dura and its compiler. Possibly, this future version can then build upon the translation we propose here as the extension may translate those extended constraints to the constraints as restricted by the current version. However, for the time being we find that it is important to offer basic complex action functionality even if it means that we have to accept some shortcomings.

3.4.3. Semantic Analysis of Executability

Before translating a complex action it is highly desirable that the compiler indicates whether the provided execution constraints will lead to issues, i.e. one of the composed actions will not be initiated, during actual execution by the runtime system. The first step of this semantic analysis is of course checking whether the provided constraints are of the allowed type constraining the initiation of an action in regard to the initiation or end of another action that occurred before.

Sometimes it is trivially obvious (to detect) that an action within a complex action will never be initiated. This is the case if it depends on the success of another action execution for which we know that we will never receive such feedback. For every action in a Dura program it may be specified when it is considered to be successful with a `succeeds on` block. When analyzing the execution constraints of a complex action one can extract all dependencies on action execution ends. If any of the associated action definitions in the Dura programs does not have a `succeeds on` block the complex action is not executable, i.e. some actions within it are guaranteed never to be executed.

Even constraints that have have been checked as described above can lead to a critical issue. Consider the following example:

- Let a and b be actions to be executed.
- Action b shall be initiated after action a has ended successfully.
- Action a shall be initiated after action b has ended successfully.

Even though we will find a translation for this in the language of Dura_C we already know that it will cause circular waiting during runtime, i.e. execution of action b waits for action a's end while execution of action a waits for action b's end.

In the example above the conflict is immediately visible but our analysis also has to detect cases where this circular waiting occurs transitively as in the following example encompassing three actions:

- Let a, b, and c be actions to be executed.
- Action b shall be initiated after action a has ended successfully.
- Action c shall be initiated at least two minutes after action b has been initiated.
- Action a shall be initiated after action c has been initiated.

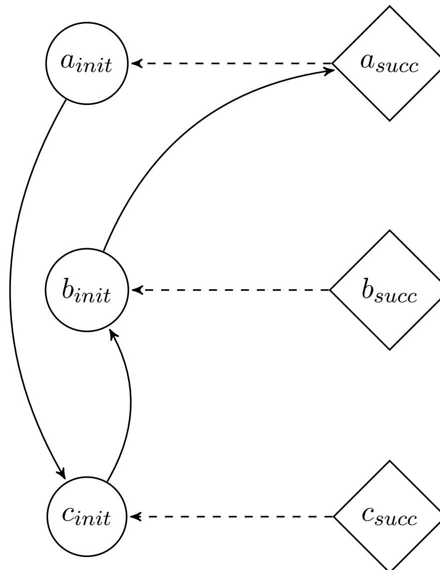


Figure 3.4.1.: Dependency graph example

The analysis we employ is based on first building a dependency graph from the constraints and then running a cycle detection algorithm on it. Due to the way the execution constraints have been restricted, initiation of an action always involves observation of a time point associated with another action execution. So, if the execution of an action x depends on a time point p (initiation or success of an action) it means that action x will always be initiated after the time point p, so x's execution depends on the detection of an event associated with that time point. As such, the dependency graph can be constructed as follows:

- The set of vertices is comprised of the initiation and (successful) end time points of all the actions within the complex action.
- For every initiation time point p one can determine from the constraints on which other time points u_i it depends. An edge $p \rightarrow u_i$ is added for every such dependency.
- Every end of an action comes after its initiation, thus the end time point depends on the action's initiation time point. For every action an edge for this dependency is added.

Figure 3.4.1 shows the dependency graph for the second example with actions a, b, and c. The dashed lines here stand for implicit dependencies (not deriving from user-provided constraints), the diamonds stand for uncontrollable, i.e. merely observable, time points, and the circles stand for controllable, i.e. we know what is required for them to occur, time points. Note that these are only distinguished in this figure, for the graph and the analysis this information is irrelevant.

In order to check for the existence of cycles, for every time point a modified depth first search can be performed starting from the respective node. If during that search the start node itself is encountered again there exists a cycle. There exists no cycle if none of those searches found a cycle. If the dependency graph of a complex action is free of cycles it is executable and guaranteed not to cause circular waiting during runtime. There are surely more sophisticated, efficient algorithms for this task. However checking is performed at compile time where the potential gain in efficiency is not as crucial for now.

In the example of figure 3.4.1 the cycle $a_{succ} \rightarrow a_{init} \rightarrow c_{init} \rightarrow b_{init} \rightarrow a_{succ}$ is found. In this case the compiler would issue an error message and not continue compilation.

Together with the initial checks for allowed constraint type and dependence on successes of actions that actually specify success, this cycle detection suffices to ensure executability of the complex action *if* all its contained action executions which need to succeed actually succeed during runtime. Note that not all actions within a complex action need to succeed for that to be the case. This is for instance the case for an action x if no action initiation depends on action x's success.

Still, semantic analysis cannot make guarantees that in any case all actions specified within a complex action are actually executed. For instance, if an action x depends on another action y's success, but during runtime the conditions required to derive success of action y are never met, x will never be executed. This follows from the inherent difficulties associated with actions in Dura. At compile time it is not known whether an action execution will actually succeed or not during runtime. It is only known whether an action will surely never succeed, i.e. when its success is not specified altogether.

In conclusion, in general it cannot be guaranteed that executing a complex action will actually cause all actions that are part of it to be executed. If a complex action passes the semantic analysis described here, it only means that for any action x within it the possibility of its initiation cannot easily be ruled out. In this regard the semantic analysis remains incomplete.

The reason for the incompleteness of the semantic analysis of executability is due to the fact that it currently does not check whether event queries within a complex action's `fails`

`on` or `succeeds on` block, together with the execution constraint would lead to guaranteed failure or no success of the complex action during runtime. For instance, if one were to constrain an action b's initiation to wait ten minutes after the end of an action a, but in the `succeeds on` block one would specify a query to the effect that a's end and b's initiation have to occur within two minutes, the complex action (with a and b) will never succeed during runtime[1]. Analyzing possibly arbitrary queries within the `fails on` and `succeeds on` blocks for consistency is not part of the semantic analysis described here, and it is outside the scope of this thesis.

3.4.4. Translation of Complex Action Rules

Complex actions come in two varieties, as (nested) composite actions in reactive rules (`ON` ... `DO` ... `END`) and as part of complex action rules (`FOR` ... `DO` ... `END`). The former can be expressed by moving these composite actions to complex action rules and calling those newly defined actions in their former place. As such, we will mainly concentrate on the translation of complex action rules, the more general construct.

Complex action rules appear as part of an action definition (in which an action and its schema are specified). They are similar in usage to user-defined functions in imperative programming languages. Disregarding actual syntax here, this construct is comprised of:

- A term defining the name of the user-defined action and the parameters it accepts.
- A block which contains terms used for executing actions with provided parameters. Any such action call in this block contains an optional identifier. This is important since one might wish to execute two instances of the same action (maybe with different parameters) but still might need to be able to reference those instances separately.
- An optional block containing the execution constraints.
- Two optional blocks for specifying success or failure of the complex action.

Translating complex action rules is based on an important feature provided by the runtime system: Execution of an action entails events that contain provenance information. When an action is initiated, ended successfully or failed, respective events are derived by the runtime system [13]. This property can be combined with reactive rules, that are an available feature of Dura_C, to form a translation of complex action rules. Consider the following example as informal complex action rule:

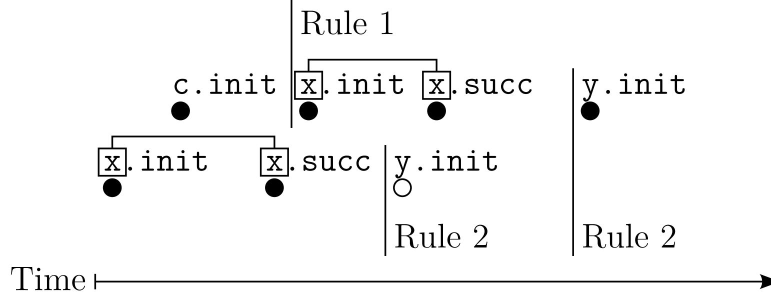
- Let `c{}` be a complex action.
- When action `c{}` is initiated, action `y{}` shall be initiated. When action `y{}` has ended successfully, action `x{}` shall be initiated.

The initial idea for translating this is to create two reactive rules reacting to the events generated by the abstract machine during runtime:

1. "Upon detection of an event stating action `c{}` was initiated, initiate action `x{}`."
2. "Upon detection of an event stating action `x{}` ended successfully, initiate action `y{}`."

[1]Similar issues pertain to generally "impossible" queries in `fails on` or `succeeds on` of actions encompassed by an action composition.

Figure 3.4.2.: Instance synchronization issue example



Although the aforementioned translation is and remains the basic approach, it is flawed if left unaltered. Consider the case that during runtime an instance (x_1) of action `x{}` is initiated and an instance (c_1) of action `c{}` is initiated. This causes another instance (x_2) of action `x{}` to be initiated. Reactive rule number two will react to both instances of `x{}` although the first instance (x_1) is entirely unrelated to the complex action specified by the complex action rule. Figure 3.4.2 shows such an example. The first firing of rule two is due to the successful end of action instance x_1, which is not related to the complex action.

Somehow we need to ensure that reactive rule number two only reacts to those instances of action `x{}` that are related to an instance of our complex action. By extending the involved actions' schemata and with the help of the provenance information (i.e. system-assigned ID, supplied parameters, etc.) provided in entailed events, the rules can be adapted as follows:

1. "Upon detection of an event stating that `c{}` with system ID `var id_c` was initiated, initiate `x{ parent-id{ var id_c } }`."
2. "Upon detection of an event stating that `c{}` with system ID `var id_c` was initiated and another event stating that `x{ parent-id{ var id_c } }` ended successfully, initiate `y{ parent-id{ var id_c } }`."

Now rule number two will react no more to unrelated events but only to those related to an instance of the complex action as intended by the complex action rule. Still, a different instance related problem may occur. Consider the following example:

- Let `d{}` be a complex action.
- When action `d{}` is initiated, action `z{ p{ 10 } }` (c) shall be initiated. When this action has ended successfully, action `z{ p{ 10 } }` (b) shall be initiated. Finally, after this action has ended successfully, action `z{ p{ 5 } }` (a) shall be initiated.

The order of the different executions must be maintained, i.e. only after two executions with parameter `10` shall the action `z{ ... }` be initiated with parameter `5`. However, a reactive rule as we would translate it until now only reacts to the detection of an event that is signaling that *any* action named `z{}` was successful. When it comes to the initiation of `z{ p{ 5 } }` there is no way to distinguish whether a detected entailed event belongs to the first `z{ p{ 10 } }` (c) or the second `z{ p{ 10 } }` (b).

Such issues of several actions with the same name as part of a complex action can be solved by giving the actions within a complex action rule unique (to the same complex action)

numbers and using them for execution and querying (as provenance information of entailed events). They can be considered static, unique identifiers for the actions within a complex action. The above complex action `d{}` would be (correctly) translated as:

1. "Upon detection of an event stating that `d{}` with system ID `var id_d` was initiated, initiate `z{ parent-id{ var id_d }, num{ 0 }, p{ 10 } }`."
2. "Upon detection of an event stating that `d{}` with system ID `var id_d` was initiated and another event that `z{ parent-id{ var id_d }, num{ 0 } }` ended successfully, initiate `z{ parent-id{ var id_d }, num{ 1 }, p{ 10 } }`."
3. "Upon detection of an event stating that `d{}` with system ID `var id_d` was initiated and another event that `z{ parent-id{ var id_d }, num{ 1 } }` ended successfully, initiate `z{ parent-id{ var id_d }, num{ 2 }, p{ 5 } }`."

The numbers serve as additional instance synchronization information that can be easily queried in reactive rules. Unlike the matching with the correct complex action instance as described earlier, the solution for the problem of reacting to the correct instance of an action within a complex rule does not even require the use of Dura variables.

Constraints asking for blocking the initiation of an action q in relation to a time point and an additional amount of time are realized by an appropriate negated query for a non-detectable event in the query part of the reactive rule for action q. The constraints to that negated query are designed so as to ensure that during runtime the querying waits the desired amount of (blocking) time during which of course the non-existing event cannot be detected. This way, blocking the initiation of an action is achieved by means of the inbuilt blocking mechanisms for query negation in Dura_C.

The optional blocks for specifying when a complex action c is successful (or fails) may contain a common Dura event query. If this query matches, the complex action is considered successful (or failed). Since complex actions are not dealt with by Dura_C or the abstract machine we have to offer this functionality by appropriate translation. This means that a complex event stating that action c was successful (or failed), by means of a deductive rule, will be added to the resulting Dura_C program.

This chapter and section is only an informal, conceptual overview and thus limited in the amount of detail described. Further information on how all the different aspects of complex actions are translated to Dura_C is found in section 4.5 within chapter 4.

3.5. Related Work

Desugaring is a widely used concept in language and compiler design, as described for instance by Turbak et al. [28]. Concrete examples of sugaring and desugaring are found, for instance, in the functional programming language Haskell [14], where e.g. lists and list comprehensions are syntactic sugar for more complicated expressions.

An example of syntactic sugar in a declarative language are definite clause grammars (cf. [27], p. 375) which are supported by many implementations of the language PROLOG, for instance GNU Prolog [11]. Definite clause grammars (DCG) rules are translated, i.e. desugared, into standard Prolog clauses.

Since Dura is a new language, related work applicable to its desugaring is not easily found or does not exist. To the best of our knowledge, the desugaring transformations described in this thesis have not been described in this exact form. We believe this is largely due to the fact that they are composed of simple ideas and steps, that are possible not too valuable to the research community.

However, Eckert's event processing language XChange^{EQ} [10] which can be seen as a precursor of Dura has given helpful insights and pointers especially in regard to understanding temporal relationships, on which the desugaring of Dura complex action rules described in this thesis relies. The static determination of so-called temporal relevance of events in this language as described in [7] uses the same temporal constraints notation as was desired for constraining the execution of actions as part of a complex action. However, the reasoning on these constraints is focused on enabling garbage collection during runtime and does not immediately translate to the planning of action execution.

Additional research brought temporal constraint networks to our attention, and therein in particular the Simple Temporal Problem (STP), both described by Dechter et al. [9]. They describe solutions to the so-called temporal constraint satisfaction problem, where constraints over time point variables are checked for consistency and possible scenarios for the variable values are calculated. In case of the STP all constraint specify only a single interval. While in the general TCSP one could specify that a time point b occurs between 30 and 40 time units after time point a or between 70 and 100 time units after time point a, in the STP only one such interval per time point pair may be specified. These intervals can be represented by two inequalities which lend themselves to a graph representation. Consistency of a given set of time point variables and constraints can then be determined in polynomial time by solving the all-pairs shortest path problem, for instance by applying the Floyd-Warshall algorithm (cf. [8], p. 630), on this graph and checking that there are no negative cycles [9, 20].

So, with this method of checking inconsistency of a STP network of constraints one can determine whether for a given set of events and temporal constraints there exists a solution in the form of time values for their occurrence, i.e. whether there is a possible scenario. For instance, let a be the time a plane leaves Munich, let b be the time the plane arrives in Tokyo, and let c be the time it arrives back in Munich. Suppose we say that it takes between 12 and 14 hours from Munich to Tokyo ($12 \leq b - a \leq 14$), and between 13 and 15 hours back ($13 \leq c - b \leq 15$). If we add the constraint that it may take between 10 and 30 hours for a round trip ($10 \leq c - a \leq 30$), there exists a scenario where all constraints are satisfied, e.g. $b - a = 12$ and $c - b = 14$. If, for instance, the round trip time has to be between 10 and 16 hours, no such scenario that satisfies all constraints can exist.

Simple temporal networks (STNs) can be considered from a more active perspective and one can build the notion of executing them, where executing a time point means assigning the current time to it, and execution as a whole has the goal to maintain the network's consistency while executing the time points[20]. This property has been used by autonomous systems as described by Muscettola et al. [18] to execute tasks as given by temporal constraints. However, mainly time points entirely under the control by an executing system are considered[2]. The actions in Dura have an end time point we have no control over and only

[2]There is some hinting in [18] at handling observable events as well.

limited knowledge about, which makes solutions to the STP insufficient as is. After all, we need to be able to make the execution of an action depend on only observable time points (like ends of actions) as well.

Vidal et al. [29] introduce the concept of controllability of the Simple Temporal Problem with Uncertainty (STPU) as opposed to the mere consistency of STPs. Solutions to determining this controllability as well as executing such STPU networks have been introduced and advanced by Morris et al. [17, 16, 15] and Stedl et al. [26]. However, these algorithms require the upper bound of contingent, i.e. uncontrollable, temporal constraint intervals to be finite. Hence, for the time being these methods cannot be applied to Dura as our actions can possibly last forever, i.e. a successful end might never be observed. Note that knowing the upper bounds on duration *would* have the advantage of enabling a system to check whether an action can successfully be executed entirely during the execution of another.

All these solutions rely on so-called dispatchers that actively execute time points and propagate changes to the temporal constraints within the constraint network, e.g. the actual duration it took between executing two time points is actually set as constraint and the changes to other constraints in the network are propagated. On account of being limited to Dura_C constructs, as is the case for this thesis, this dispatching and subsequent updating of constraints during runtime is hard to realize. However, the most damning issue with all these solutions that prevents us from using these methods for planing complex actions in Dura_C is, that already for executing controllable time points (as in STNs) reliably one needs to know about the executing system's latency properties and take them into account when propagating constraints during execution by a dispatcher [18]. This is currently not possible for us, since the runtime system of Dura does not provide this information, especially not at the level of Dura_C.

In the following we describe other work related to the task of action planning. Ziparo et al.[30] employ a Petri net representation for multi-agent plans in which one can express action failures, loops, concurrency, and action synchronization. This approach appears to be much more expressive than what can currently be specified for complex actions in Dura.

Given the fact that Dura_C offers reactive rules, the so-called reactive planning paradigm appears to be a viable path for translating Dura's complex actions. In reactive planning, "no specific sequence of actions is planned in advance [...] instead of producing a plan with branches, [a planner] produces a set of condition-action rules" [22]. For instance, Schoppers [23] introduces so-called universal plans which describe the execution of actions depending on conditions derived from sensory feedback. However, execution of these universal plans relies heavily on this continuous feedback which, in case of delays, can lead to uncertainty in how best to continue execution [23].

The aforementioned approaches are relevant to complex actions in Dura but are either overly expressive, too constrained, or offer only limited methods for checking soundness. For the complex actions as they are described in this thesis these approaches were not further pursued. However, they gave valuable insight and might proof helpful for the future translation of extensions (e.g. conitional statements) that could not be covered in the scope of this thesis.

4. Implementation and Algorithms

The implementation of the general ideas introduced in chapter 3 in the form of transformations on an abstract syntax tree deserves in-depth treatment. This chapter is a detailed description on what types of transformations were devised, how they were designed, and how they build upon each others' results.

4.1. Proof of Concept

As part of this thesis a proof of concept desugarer for Dura was created the implementation language of which was Java SE 6. For lexical and syntactic analysis we employed ANTLR v3, an LL(*) parser itself implemented in Java [21]. Before work on the actual implementation started, a testing mechanism was devised in the form of an ANTLR tree parser that validates whether the input AST is of the type that the Dura_C compiler accepts. This AST validation parser was then extended to a pretty printer for such ASTs to be used for checking results and debugging of the desugarer prototype.

The prototype directly builds upon the translation steps described hereafter. However, a description of the exact implementation details such as source code, class structure, and others does not lend itself to an easy understanding of the core of the translation from Dura to Dura_C. This is why these minutiae are omitted from this chapter. Instead, the transformations are described on a conceptional, yet detailed, level independent of software implementation specifics. Where applicable, detailed implementation considerations are pointed out and addressed.

4.2. Semantic Analysis and Typing

The input to the desugarer is presumed to be free of syntactic errors as well as statically detectable semantic errors such as those regarding typing and scope. If we needed semantic analysis only for error detection we could safely go on from here since we know that by the time desugaring is performed, the code has already been checked. However, some of the transformations described in this chapter need to construct new schemata for events and actions, based on type information to be found in the constructs to be transformed.

The fact that semantic analysis has not been implemented at the time of the prototype's creation (but was to be implemented in the near future) was worked around by implementing only the bare minimum functionality for determining the type of a variable or term leaf in general. In an initial step the Dura AST is walked and from it schemata as well as custom type definitions are collected and amended by implicit schema terms for IDs, reception times, and others, depending on the type of schemata (event, stateful object, action).

When, during transformation, we encounter a term leaf and wish to determine its type according to the collected schemata we can do so by comparing it with its corresponding schema term.

Listing 4.2.1: Term

```
a{
  b{
    d{ x }
  }
}
```

Consider a term for the object `a` presented in listing 4.2.1 that contains an expression x. The type of x to be found as term leaf can be determined by first walking up the term hierarchy until the term's root is found and save the sequence of passed term labels. In this example this would be `d`, `b`, `a`. Knowing that the root is named `a` we can look this up in the data structures built by the initial AST walking and find its schema. Let its schema be the one presented in listing 4.2.2

Listing 4.2.2: Schema term

```
a{
  b{
    c{ string },
    d{ int }
  },
  e{ float }
}
```

The schema term can now be traversed in the order given by the reversed term label sequence, i.e. `a`, `b`, `d`. This reveals that x must be of type `int`.

Listing 4.2.3: Schema term (with custom type)

```
a{
  b{ cd },
  e{ float }
}
```

Assume we encountered a custom type `cd` as (schema) term leaf below `b` (listing 4.2.3). In this case we would just have to find the definition of `cd` and continue the hierarchy sequence descent therein. Note that the custom type definitions are added to an easily and efficiently accessible data structure (e.g. a hash map between the name of a type and its definition) in the initial preparation step just like the schemata.

Yet another case could occur. Namely, the schema term could look like the one presented in listing 4.2.4. Following the hierarchy sequence until its end `d` finds that this end does not contain a type definition but further terms instead. The result of this would be that x is found to be of composite type (or schema) `< e{ int }, f{ int } >`.

Listing 4.2.4: Schema term

```
a{
  b{
    c{ string },
    d{
      e{ int },
      f{ int }
    }
  },
  g{ float }
}
```

Since we assume that the provided AST is type correct this method should always find a result. However, in the actual prototype implementation such erroneous cases are in fact also considered and their occurrence causes abortion of the translation. Also, for this method some amount of additional bookkeeping is required, namely whenever during the desugaring process a new event, stateful object, or action schema is constructed and added to the AST it needs to be added to the semantic analysis data structures. Of course this only needs to be considered in those desugaring transformations that actually cause addition of new schemata or change of existing ones.

4.3. Transformation Sequence Overview

The order of applying the desugaring transformation, described in this chapter, to the AST is crucial and has been exploited in their design so as to limit the need for rewriting code. The following brief description over what each single transformation does focuses on this aspect and is meant to be revisited in case the reader needs further overview not provided within the detailed transformation sections themselves.

1. *Resolving Nested Action Compositions*: Nested action compositions found in complex action rules or reactive rules' execution parts are extracted and turned into new action definitions (with a complex action rule), i.e. complex actions. Where the nested action compositions originally appeared the new actions are called instead. After this transformation all complex action rules and reactive rules' execution parts contain only a flat list (one level) of actions to be executed.
2. *Translating Complex Action Rules*: The action composition part of complex action rules is translated into reactive rules while `succeeds on` and `fails on` blocks are translated into event definitions (with a deductive rule), i.e. complex events. After this transformation there exist no more complex action rules in the AST.
3. *Simplifying Reactive Rules*: Reactive rules are simplified as follows. Their execution part remains largely unaltered safe for substitutions of arithmetic expressions or identifiers with variables. With these substitutions in mind the query part is extracted and turned into a new event definition. Instead of the old query this complex event is now queried in the query part of the reactive rule. By doing so, handling of arith-

metic expressions in the execution part can be deferred to a later transformation that handles expressions in deductive rule heads.

4. *Resolving Nested Event Compositions*: Nested event compositions are extracted and turned into new event definitions, i.e. complex events. Where they originally appeared the new events are queried instead. After this transformation all deductive rules contain only a flat list (one level) of events to be queried.
5. *Translating Existential Quantification*: Existential quantification constructs (`exists`) are turned into groupings.
6. *Replacing Aggregation Operations*: For deductive rules, if the head contains a `group by` block it is moved to the end of the body. Aggregation operations in the head of rules or within a `where` or `let` block in the body are moved to the closest existing `aggregate` block or a new one.
7. *Resolving Expressions in Rule Heads*: In the head (term) of rules, any term leaf not being a variable or identifier is substituted with a variable. These variables are defined (as the expressions) in a `let` blocks that is added to the end of the corresponding rule's body.

The following sections cover these transformations in the same order they have to be applied, i.e. as they are presented in the list above. We believe that this way the process of transformation as a whole from a Dura AST to a Dura_C AST can be traced more easily than by ordering the sections differently.

4.4. Resolving Nested Action Compositions

In Dura action compositions may occur arbitrarily nested within other action compositions. This means that they may occur where in Dura$_C$ only named action executions may be called. Removing these nested action compositions considerably facilitates all the following transformations dealing with complex actions and reactive rules and is very similar in nature to the transformation for resolving nested event compositions (cf. section 4.7).

4.4.1. Nested Action Compositions

Nesting of action compositions may occur both in complex action rules as well as the execution part, i.e. the head, of reactive rules. A nested complex action is not associated with a name just like the top-level action composition in a reactive rule. A reactive rules' execution part may be an action composition that constraints execution of its contained actions as opposed to one that does not, i.e. a `concurrent` block without supplement. Such non-trivial action composition can be considered as a single nested action composition within a trivial concurrent action composition.

Let $E = \{exec_1, \ldots, exec_m\}$ be a set of action execution calls and let $supplement_E$ be the supplement to the composition encompassing E. This supplement may contain a `where` block for constraining the executions, a `succeeds on` block, and a `fails on` block both for specifying when the complex action is successful or fails. If this supplement is empty we are dealing with a concurrent action composition of the actions in E, i.e. a trivial action composition. The exact shape (or content) of $supplement_E$ is of no particular interest for the transformation described in this section. Listing 4.4.1 shows the general action composition pattern using the definitions introduced above.

Listing 4.4.1: Action composition pattern

```
concurrent {
  exec_1 ,
  ⋮
  exec_i ,
  ⋮
  exec_m
} supplement_E
```

The action execution calls in E may be atomic (or flat) or contain an action composition i.e. a nested action composition. In each resolution step of the transformation described in this section only a single nested complex action is resolved. It suffices to consider such a single nested action composition without regard to the exact nature of the other action execution calls in E.

Let $E_a = \{exec_{a_1}, \ldots, exec_{a_n}\}$ be a set of action execution calls and let $supplement_E$ be the supplement to the composition encompassing E_a. Then let $exec_i$ be the action execution call containing the nested action composition shown in listing 4.4.2.

Listing 4.4.2: Nested action composition in `action` a

```
action a: concurrent {
  exec_a1,
  ⋮
  exec_an
} supplement_Ea
```

Listing 4.4.3 shows the expanded execution call $exec_i$ within the outer composition, described earlier, encompassing E. The general idea for resolving this, i.e. transforming a nested action composition into a flat action execution call, is to first name the nested action composition and turn it into a new complex action by means of a new action definition with a complex action rule. Subsequently, the original action execution call $exec_i$ is substituted with an atomic (by name) action execution call referring to the action defined by the newly created complex action definition and rule.

Listing 4.4.3: Nested action composition in outer composition

```
concurrent {
  exec_1,
  ⋮
  action a: concurrent {
    exec_a1,
    ⋮
    exec_an
  } supplement_Ea,
  ⋮
  exec_m
} supplement_E
```

4.4.1.1. Handling Non-Trivial Parameters

Resolution of nested action compositions is complicated by the fact that in Dura the parameters for action execution calls are not necessarily variables but may also potentially be arithmetic expressions or identifiers[1]. However, Dura$_{\mathrm{C}}$ does not accept arithmetic expressions as parameters.

Listing 4.4.4 presents such a scenario with a complex action rule. When extracting the nested action composition, arithmetic expressions can be dealt with in two different ways:

- By finding out which variables are used within the expressions, only expecting those variables as input in a new complex action rule, and handing them over as parame-

[1]Identifiers can only occur as parameters in the execution part of reactive rules but not in complex action rules.

ters where the original action execution call with nested action composition has been substituted with an action execution call to the new complex action.

- By replacing every non-variable parameter with a unique variable and remembering what parameter belongs to which variable. In the new complex action rule created from the nested action composition only those variables are used. However, in the flat action execution call the nested action composition has been substituted with, the original expressions are handed over as parameters.

Listing 4.4.4: Expression in parameter example

```
FOR
  action a: v{ w{ var A } }
DO
  concurrent {
    action b: x{}
    action c: concurrent {
      action d: y{ p{ var A } },
      action e: z{ q{ var A * 2 } }
    }
  }
END
```

While the first method works for transforming complex action rules it is insufficient for reactive rules as those could contain identifiers (e.g. `event a`) as parameters. This is why the second, more general method was chosen for the transformation described in this section. It is best illustrated by an example translation of the complex action rule of listing 4.4.4.

Listing 4.4.5: Example translation (new complex action)

```
ACTION
  schema
WITH
  FOR
    _0_0{
      _0_0_0{ var _0_0_0 },
      _0_0_1{ var _0_0_1 }
    }
  DO
    concurrent {
      action d: y{ p{ var _0_0_0 } },
      action e: z{ q{ var _0_0_1 } }
    }
  END
END
```

Listings 4.4.5 and 4.4.6 show such an example translation. The action schemata shall not be considered here and are not described in detail. The schema of action `_0_0` (*schema*) can be constructed from the schemata of actions `y` and `z` which are not defined in this (incomplete) code snippet.

Listing 4.4.6: Example translation (changed composition)

```
ACTION
  ...
WITH
  FOR
    action a: v{ w{ var A } }
  DO
    concurrent {
      action b: x{},
      action c: _0_0{
        _0_0_0{ var A },
        _0_0_1{ var A * 2 }
      }
    }
  END
END
```

A naming pattern of using an underscore character as is used in this example is actually employed in the prototype implementation. The reason for this is that the lexical and syntactic analysis forbids such names and it can be guaranteed that programmers have not used names with underscores as initial character. Additionally, by using a naming pattern like this instead of just giving random names, additional information can be encoded. This information is a tremendous help for debugging. For instance, one can immediately see which names have been created by the desugarer and by which particular transformation therein.

4.4.1.2. Resolution

Let $varToPar$ be a mapping between variables and parameters, and let the following functions (the implementations of which are fairly trivial) be provided:

- Parameters shall take an AST and return references to all the parameters, i.e. action term leaves found in a given action term or action composition. This can be implemented by a depth (or breadth) first search that collects all nodes in the AST that are not terms themselves but have a term as their parent. Note that this function collects all parameters even those in further deeply nested action compositions.
- Create-Variable shall create and return a new uniquely named variable.

The function Build-VarToPar defined in algorithm 4.4.1 builds such a mapping between variables and parameters for those parameters for which it is required and replaces the parameters in the given action composition's AST with newly created variables.

Recall listings 4.4.1 and 4.4.3. Assuming that algorithm 4.4.1 has been applied to $exec_i$ and a variable to parameter mapping $varToPar_{exec_i}$ has been built, $exec_i$ can then be resolved as follows. Let $label$ be a new unique label name, let $label_{v_1}, \dots, label_{v_k}$ be unique label names, and let $v_1, \dots, v_k$ be all the variables in the domain (or key set) of $varToPar_{exec_i}$. Let $type_{v_1}, \dots, type_{v_k}$ be the types of these variables which can be inferred by the occurrence of

Algorithm 4.4.1: Building variable to parameter mapping
BUILD-VARTOPAR(*actionComposition*)
(1) Initialize mapping *varToPar*
(2) **foreach** $p \in$ PARAMETERS(*actionComposition*)
(3) **if** p is a variable
(4) **if** *varToPar* does not contain (p, p)
(5) Add (p, p) to *varToPar*
(6) **else**
(7) $v \leftarrow$ CREATE-VARIABLE()
(8) Add (v, p) to *varToPar*
(9) Replace p by v (in the AST)
(10) **return** *varToPar*

the variables within the action terms of action execution calls. For performance reasons, in the actual prototype implementation this is done while building the $varToPar_{exec_i}$ mapping.

Listing 4.4.7: Nested action composition in `action` a as named complex action

```
ACTION
  label{
    label_v1{ type_v1 },
    ⋮
    label_vk{ type_vk }
  }
WITH
  FOR
    action a: label{
      label_v1{ v1 },
      ⋮
      label_vk{ vk }
    }
  DO
    concurrent {
      exec_a1 ,
      ⋮
      exec_an
    } supplement_Ea
  END
END
```

Listing 4.4.7 shows the action composition that was previously nested in `action` a as part of a uniquely named complex action. Recall that the elements of $E_a = \{exec_{a_1}, \dots, exec_{a_n}\}$ might have changed due to potential replacements in the AST by algorithm 4.4.1. For a variable v_i let p_i be the corresponding parameter found in $varToPar_{exec_i}$ i.e. $(v_i, p_i) \in varToPar_{exec_i}$. Finally, subsitution of $exec_i$ in the original composition yields listing 4.4.8.

In summary, the steps described above (handling parameters and building the variable to parameter mapping, creation of a new rule, and substitution) describe the resolution of

Listing 4.4.8: Action execution call $exec_i$ substituted

```
concurrent {
  exec_1 ,
  ⋮
  action a: label{
    label_v1 { p_1 },
    ⋮
    label_vk { p_k }
  },
  ⋮
  exec_m
} supplement_E
```

a single nested complex action within an outer action composition to be found within a reactive rule or complex action rule.

4.4.2. Action Composition in Reactive Rules

The transformation described in this section also resolves non-nested, i.e. top-level, non-trivial action composition in reactive rules. These are handled in this section because, as was indicated earlier, such cases can be related closely with nested action composition. The only shape such a non-trivial action composition can have is presented in listing 4.4.9.

Listing 4.4.9: Non-trivial action composition in reactive rule pattern

```
ON
  ...
DO
  concurrent {
    exec_1,
    ⋮
    exec_i,
    ⋮
    exec_n
  } where {
    constraints_E
  }
END
```

Let $constraints_E$ contain execution constraints for the execution calls in E. Without the `where` block this action composition would be trivial in the sense that firing of the associated reactive rule during runtime would merely cause (semi-)concurrent execution of the contained actions without further specification. Note that Dura_C only accepts trivial action composition.

Listing 4.4.10: Nested (non-trivial) action composition

```
ON
  ...
DO
  concurrent {
    action: concurrent {
      exec_1,
      ⋮
      exec_i,
      ⋮
      exec_n
    } where {
      constraints_E
    }
  }
END
```

The action composition of listing 4.4.9 is easily resolved by turning it into a nested action composition as shown in listing 4.4.10. After this, the same steps as in section 4.4.1.2 can be

employed. In the actual prototype implementation this has been optimized to be handled in one step but the principal remains the same as described here.

4.4.3. Complete Transformation

The previous (sub-)sections describe how a single action composition is resolved. Although this is the main challenge the whole transformation's explanation is incomplete without a description on how it is applied to an entire program's AST. The following steps outline this complete transformation:

1. Collect all reactive rules with a non-trivial action composition as their execution part and resolve the action composition as described in section 4.4.2.
2. Collect all top-level action compositions found in complex action rules and reactive rules.
3. Consider every nested action composition found at the top level of these action compositions and resolve it as described in section 4.4.1.2. Keep track of the newly created complex actions and for the action composition in their complex action rule do this same step (step three) again.

As can be seen in step three the complete transformation is performed top-down. It stops when no more nested action compositions are found.

4.5. Translating Complex Action Rules

Complex action rules are the crucial core of complex action definitions in Dura. After application of the transformation described in the previous section, non-trivial action compositions (including execution constraints, etc.) only occur as part of complex action rules. There are also no more nested action compositions to be found anywhere. This facilitates the translation of complex actions immensely, which is important as the translation of complex action rules is arguably the most complicated desugaring transformation in this chapter.ds dd This is largely due to the many parts complex actions contain that need to be handled specifically. The general idea of translating complex action rules as introduced in section 3.4 and further details are elaborated on in the following.

4.5.1. Complex Action Rules

Complex action rules in Dura occur only as part of complex action definitions (`ACTION` ... `WITH` *complexActionRule* `END`) and specify a user-defined action much like function or procedure definitions do in imperative programming languages. After a complex action rule has been translated by the desugarer it needs to be removed from the surrounding action definition as Dura_C does only accept such action definitions without complex action rule (`ACTION` ... `END`).

Let $identifier_p$, $identifier_1$, ..., $identifier_n$ be unique identifiers, let $term_p$ be the action term for the action to be specified, let $term_1$, ..., $term_n$ be action terms of other user-defined, inbuilt internal, or external actions, and let $supplement_E$ be the supplement to the action composition encompassing the action execution calls `action` $identifier_i$`:` $term_i$, ..., `action` $identifier_n$`:` $term_n$.

Listing 4.5.1: Flat complex action rule pattern

```
FOR
  action identifier_p: term_p
DO
  concurrent {
    action identifier_1: term_1,
    ⋮
    action identifier_n: term_n
  } supplement_E
END
```

This supplement may contain at most one `where` block, at most one `succeeds on` block, and at most one `fails on` block. It may also be empty, i.e. not provided at all, which would make translation of the rule a rather simple matter. Listing 4.5.1 shows the general pattern for a so-called flat complex action rule. We only need to consider such flat rules, i.e. ones that only contain action execution calls with action terms, since at the time of applying the transformation described in this section, the one applied prior to it has already resolved all nested action compositions.

4.5.2. Handling Execution Constraints

Using the aforementioned definitions, let *constraints* be a conjunction of execution constraints and let `where {` *constraints* `}` be a part of $supplement_E$. Without loss of generality, the elements of *constraints* are temporal constraints (cf. [7] and [9]) of the (limited) form $x - y < -d$ or $x - y \leq -d$ with time points x and y and a positive duration (natural number and time unit) d.

Within complex action rules there are only two possible ways to express a time point, either by `begin(action` *identifier*`)` or `end(action` *identifier*`)` for an action identifier *identifier*. The former refers to the initiation of an action and the latter to the successful end of an action in a complex action rule with the given identifier. Note that an identifier may not be confused with an action's name. An identifier always refers to an instance of an action execution as part of a complex action rule. Every identifier in a complex action rule has to be unique, whereas action names, i.e. top labels of action terms, do not need to be unique. This allows for executing the same action several times, possibly with different parameters. The action identifiers can be used to relate these different executions.

4.5.2.1. Controllable and Uncontrollable Time Points

There is an important qualitative distinction to be made regarding time points. Following the definitions found in [17], [16], and [15], *controllable* time points are those time points that the runtime system has control over, while *uncontrollable* time points are those that can only be observed but cannot be controlled.

In regard to the pattern described above (cf. listing 4.5.1) the time point of initiating an action *within* the composition are controllable whereas the respective end time points are uncontrollable. This is due to the fact that we can only initiate actions but cannot preemptively tell an action when to successfully end. This is something the action itself decides per its implementation and definition during its execution and is highly dependent on what actually happens during runtime.

For the action to be specified by a complex action rule (`action` $identifier_p$`:` $term_p$) the opposite is true. From the point of view of the complex action rule itself we cannot know when it is going to be initiated, we can only observe it. On the other hand, we do have *some* influence on when it successfully ends. For instance, one might specify that a complex action only ends successfully after all the actions within the composition have ended successfully. This aspect is not covered here but revisited in section 4.5.5.

Note that, of course, we do not know the exact occurrence times of controllable time points beforehand, but we can control, i.e. plan, them in relation to other time points which is not possible for uncontrollable ones.

4.5.2.2. Limitations on Constraints

With the distinction between controllable and uncontrollable time points in mind, the types of temporal constraints allowed in *constraints* are specified as follows. Let c, c_a, and c_b be controllable time points, let u be an uncontrollable time point, and let the references to

action instances within them be pairwise distinct. This pairwise distinctiveness is necessary to prevent expressions like `begin(action` x`) - begin(action` x`) <= 2 min`. Table 4.5.1 lists the allowed temporal constraint types for *constraints*. These types of execution constraints suffice in order to plan the execution of a controllable time point, i.e. the right subtraction operand, relative to some other controllable or uncontrollable time point, i.e. the left subtraction operand. Additionally, the value on the right hand side of the inequations controls a minimum waiting time d between the two time points. When we speak of executing a controllable time point we are referring to the initiation of an action.

Table 4.5.1.: Allowed constraint types for planning

Constraint	Interpretation for planning
$c_a - c_b \leq -d$	c_b "waits" for c_a and at least duration d
$c_a - c_b < -d$	c_b "waits" for c_a and more than duration d
$u - c \leq -d$	c "waits" for u and at least duration d
$u - c < -d$	c "waits" for u and more than duration d

Consider two actions with identifiers a and b respectively. If we wish to specify that the execution of action b has to wait for action a's completion we can do so by adding the constraint `end(action` a`) - begin(action` b`) <= 0 ms`. Now suppose we want another action with identifier c to be initiated at least six minutes after action b was initiated. We can express this with the constraint `begin(action` b`) - begin(action` c`) <= -6 min`. Let $identifier_p$ be the identifier of the complex action to be specified. If the initiation of action a shall wait until at least 20 seconds after (the encompassing specified) action (identified by) $identifier_p$ was initiated we can express this with the constraint `begin(action` $identifier_p$`) - begin(action` a`) <= -20 s`.

Consider the constraint `begin(action` a`) - begin(action` b`) <= 30 s` which would be interpreted as meaning that initiation of action a occurs at most 30 seconds after action b was initiated, but action a could also be initiated at any time before action b is initiated. So, either the initiation of b depends on a past initiation of a, or the initiation of a depends on a past initiation of b but has to occur within a given time frame. The latter case causes issues as is explained later, so we would automatically have to decide for the former case. If this is the case we can require the programmer to specify it directly (e.g. with the constraint `begin(action` a`) - begin(action` b`) <= 0 s`) in the first place. As we want to avoid ambiguity for the time being, we decided to disallow such types of ambiguous constraints in *constraints*. However, for checking if an execution happened as desired during runtime such types of constraints might be important. This checking would have to be treated separately from the constraints in *constraints*. Currently this can be done by explicitly querying in `succeeds on` and `fails on` blocks.

One could add to `begin(action` a`) - begin(action` b`) <= 30 s` the constraint `begin(action` b`) - begin(action` a`) <= 0 s`. Now the order of execution is unambiguous but yet another problem awaits. At runtime, action initiation shall always be attempted at the earliest possible time, respecting the given execution constraints. However, upper bounds on execution cannot be enforced as during runtime we have no estimate on when exactly (in real time) an event stating success of an action b can be detected, processed,

and a subsequent initiation of action a depending on that outcome (success of action b) can happen. Currently, the runtime system cannot make guarantees on whether initiation actually happens on time. Hence, only actually enforceable types of constraints like those of table 4.5.1 are allowed in *constraints*.

If we were able to employ techniques similar to those used for the so-called Simple Temporal Problem with Uncertainty (STPU) [17, 16, 15] to construct a plan (if possible) from various constraints, general constraints could be allowed in *constraints*. However, it seems that these techniques would require a dynamic dispatcher with constraint propagation and thus might not allow for a static translation to Dura_C. In addition, the abstract machine and in extension the event processing system would have to provide certain real-time guarantees and the nature of Dura actions might need to be restricted in some ways.

At least for the time being (i.e. in this version of the prototype) we decided against such techniques in favor of a more straightforward translation as well as simpler, deterministic semantics.

4.5.2.3. Futile Wait Detection

For some actions it is not specified when their execution is to be considered successful. If the definition of an action without complex action rule contains no `succeeds on` block, or if the complex action rule within the definition of an action (with complex action rule) contains no `succeeds on` block, the action's success is not specified. Either during the initial semantic analysis phase or initially within the transformation described in this section, information on which actions have their success specified or not can be collected.

Let `action` a`:` $label_a$`{ ... }` be an action execution call within the action composition of the given complex action rule. If any of the constraints *constraints* contains the time point `end(action` a`)` and the action named $label_a$ does not specify when its execution is to be considered successful, compilation is aborted with an error message.

Due to the limitations imposed on execution constraints, if `end(action` a`)` is part of a constraint in *constraints* it means that some action x's initiation depends on the successful end of the action execution instance identified by a. But if we know that a's associated action named $label_a$ does not specify its success, success of a will certainly never be detected. This means that action x will never be initiated because its waiting for the successful end of a is futile.

If because of the execution plan described by the provided constraints there is at least one action within an action composition that is known to never be initiated the plan is not realizable. In this case, we already know at compile time that there is at least one action that will not be executed although it is listed within the action composition. This is an undesirable situation a programmer needs to be made aware of.

This check for guaranteed futile waiting does not exclude that an action initiation during runtime waits forever for the successful end of an action execution. However, this is nothing we can easily detect or treat during compile time.

4.5.2.4. Circular Wait Detection

The aforementioned limitations imposed on execution constraints do not automatically imply that the execution plan they describe can be realized. This is the case when during runtime the initiation of the actions in this plan would lead to circular waiting.

Listing 4.5.2: Non-executable plan

```
FOR
  action w: g{}
DO
  concurrent {
    action x: a{},
    action y: b{}
  } where {
    end(action x) - begin(action y) < 0,
    end(action y) - begin(action x) < 0
  }
END
```

Listing 4.5.2 shows a complex action rule with execution constraints that lead to a non-executable plan. Action `y` needs to wait for the end of action `x`, but action `x` will not be initiated until action `y` has ended.

Such circular waits can be detected statically by checking for cycles in a dependency graph stemming from the dependencies found in the execution constraints. As such, the first step of circular wait detection is the construction of such a graph. A single constraint $x - y < d$ or $x - y \leq d$ states that y has to occur after x has been observed and as such y depends on x. The right hand side of the inequation, i.e. the duration d, is only a secondary feature and irrelevant for the dependency fact itself.

In order to build the dependency graph one has to consider all the possible time points in the given complex action as the set of vertices V. The edges relation E can be created as follows:

1. E is empty.
2. For every action execution call (in the complex action rule) identified by i add an edge `end(action` i`)` $\rightarrow$ `begin(action` i`)` to E. This is a natural dependency as an action execution can never end before it has been initiated.
3. Group all constraints by their right time point operand and then analyze all these groups.
4. Let y be the right time point operand for such a group:
   a) For every constraint $x - y \leq -d$ in that group add the edge $y \rightarrow x$ to E. Also add the triple $(x, \leq, -d)$ to a separate list associated with y.
   b) For every constraint $x - y < -d$ in that group add the edge $y \rightarrow x$ to E. Also add the triple $(x, <, -d)$ to a separate list associated with y.

The dependency graph $D = (V, E)$ derives from these steps. With the additional lists collected during edge creation a function called Depends-On can be created. For a given time point t this function shall return a list of dependency triples consisting of time point, relation, and negative duration. While not relevant for circular wait detection, the function Depends-On will be used later during the translation of complex action rules to reactive rules.

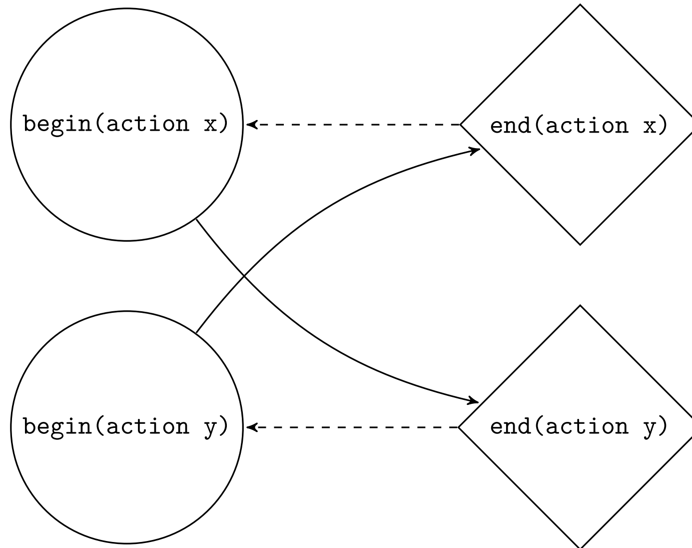


Figure 4.5.1.: Dependency graph example

Figure 4.5.1 shows the dependency graph created from the complex action rule in the non-executable plan example of listing 4.5.2. Note that despite the different vertex and edge styles in this figure, they are all represented the same in the dependency graph. When illustrating more complex examples one should try to find appropriate abbreviations instead of `begin(...)` and `end(...)`. However, in this small example we found it more important to stress the connection to listing 4.5.2 than to minimize the illustration.

A given plan deriving from execution constraints is only realizable if the dependency graph built from the execution constraints does not contain any cycles. Since the dependency graph is not necessarily connected, we have to check for every vertex v whether it is transitively connected with itself. This can be done by performing a slightly modified depth-first search (cf. depth-first search in [8] p. 540 - 541) starting from vertex v. The depth-first search algorithm is modified so as to abort when v is visited a second time and return that a cycle was detected. This proposed modification is rather simple and could possibly be optimized. However, a discussion on best choice of cycle detection algorithm for this problem is not the focus of this chapter.

4.5.3. Translation to Reactive Rules

Knowing of the temporal dependencies, the action execution calls in a complex action rule can be translated into reactive rules. However, to do so certain assumptions and require-

ments have to be made regarding the runtime system. With these requirements a basic pattern for translation can be created on which further translation steps regarding dependencies can build.

4.5.3.1. Events Entailed by Actions

Translation to reactive rules is only possible if as prescribed by Dura the abstract machine generates special events entailed by action executions. This is a core feature of the runtime system, also depended on by the implementations of external actions, without which it would be entirely impossible to translate complex action rules into reactive rules or any other existing Dura_{C} construct for that matter.

The following events are entailed by action execution:

- The event $label_a$`$initiated{ ... }` is generated when the execution of an action a (with name $label_a$) has been requested, i.e. it has been initiated.
- The event $label_a$`$succeeded{ ... }` is generated when an initiated action a (with name $label_a$) has ended successfully.
- The event $label_a$`$failed{ ... }` is generated when an initiated action a (with name $label_a$) has failed.

It shall be maintained that all of these events only occur once per execution instance. For initiation this is trivial since the abstract machine is in charge of this. Success and failure of actions is a little bit trickier and is handled in section 4.5.5. Ideally these events should be associated with time points, i.e. their begin and end reception times are equal. However, in practice this can only be maintained for action initiation events. This is also covered in section 4.5.5.

These events alone would only aid in detecting that *some* execution instance of an action named $label_a$ was initiated, succeeded or failed. Possibly, there could be any number of concurrently active execution instances of this action at a given time stemming from different execution requests. Hence, further information than just the occurrence of events entailed by action execution is necessary for controlling and testing the execution of action instances themselves.

Let $label_a$`{` $term_{p_1}$ `, ... ,` $term_{p_n}$ `}` be the action term when initiating an action a, with $term_{p_1}, \ldots, term_{p_n}$ being the provided parameter terms. In addition to the implicit elements of Dura events in general (identifier, reception time, etc.), the events entailed by action execution contain a so-called payload which contains provenance information regarding the action execution instance. The payload of an action a is the same for all three types of entailed events. It is a subterm in the respective entailed event with label `payload` and it has the form presented in listing 4.5.3.

The term leaf ID_{action} is an ID provided to an action execution instance by the abstract machine when an action execution is requested, *timeStamp* is the time stamp storing when initiation happened. When during runtime an action execution is requested the entailed ...`$initiated` event is generated by the runtime system. In case of an internal inbuilt action the runtime system then executes the action directly. If it is an external action a

Listing 4.5.3: Payload term

```
payload{
  id{ ID_action },
  initiation-time{
    begin{ timeStamp }
  },
  term_p1 ,
  ⋮,
  term_pn
}
```

so-called actuator needs to react to the initiation and runs an implementation of the action [5, 13]. In any case the abstract machine will always provide a new ID, save provenance information regarding the execution, and inform the event processing system by generating the ...`$initiated` event.

4.5.3.2. Basic Reactive Rule Pattern

With this in mind, recall the definitions and pattern for complex action rules introduced in listing 4.5.1. Translation of an action execution call `action` i: *label*`{` $term_{par_1}$ `,` ..., $term_{par_m}$ `}` within the complex action to a reactive rule starts with a simple pattern. Let $label_p$ be the label of term $term_p$ (i.e. the complex action's name) and let $parent_{ID}$ be a uniquely named variable. Listing 4.5.4 shows the basic initial pattern for a reactive rule resulting from translating this action execution call.

One name yet to be introduced is $index_i$. This is the index of the action execution call within the `concurrent` block of the complex action rule. In principal this is identical to the child index of the syntactic action execution call node under its parent, the `concurrent` block node, within the AST. It is a constant value used for ensuring that with this rule we are describing when and how an action belonging to this particular action execution call was initiated. It is later required to distinguish between different action execution instances and crucial to proper synchronization of these action execution instances.

Assuming for now that the considered action execution call `action` i: *label*`{` $term_{par_1}$ `,` ..., $term_{par_m}$ `}` has no dependencies per provided constraints, this pattern already suffices and we are finished with the translation for this action execution call. Before we further build upon this pattern its function and purpose deserves some explanation.

Even if the action execution call does not depend on other action executions, it is bound to the complex action specified by the complex action rule for the action with name $label_p$ and depends on its initiation. This is expressed in the query part of the reactive rule in the event $identifier_p$`_begin`, where its initiation is queried.

The action execution ID that can be acquired from this query takes the role of a reference to an action execution instance *inst* encompassing all action execution instances deriving from the calls within the complex action rule. In this regard *inst* plays the role of a parent

Listing 4.5.4: Reactive rule pattern

```
ON
  and {
    event identifier_p_begin: label_p$initiated{
      payload{
        id{ parent_ID },
        term_p1,
        ⋮,
        term_pn
      }
    }
  }
DO
  concurrent {
    action: label{
      _composition{
        child-index{ index_i },
        parent-action-id{ parent_ID }
      },
      term_par1,
      ⋮,
      term_parm
    }
  }
END
```

to all contained action execution instances, the so-called children action execution instances of *inst*. That is the reason for the existence of the variable $parent_{ID}$.

Together with the child index $index_i$ introduced earlier the action execution call `action` i: *label*`{` $term_{par_1}$`,` ...`,` $term_{par_m}$ `}` can be modified as seen in the reactive rule (cf. listing 4.5.4). This modification ensures that every action execution instance of the action with name *label* originating from a firing of this reactive rule will contain provenance information on what its parent (complex action) execution instance is and to which action execution call in the complex action rule it belongs.

All variables occurring in terms $term_{par_1}, \ldots, term_{par_m}$ are required to occur in $term_{p_1}, \ldots,$ $term_{p_n}$. By adding the latter to the query it can be ensured that the the execution part of the reactive rule pattern has access to all required parameter values.

4.5.3.3. Considering Dependencies

The basic reactive rule pattern introduced above only suffices to express execution of an action depending only on the complex action's initiation. This is enough if no execution constraints were provided but if constraints were provided the dependencies encoded therein also have to be considered and expressed in the reactive rule pattern.

Assuming that semantic analysis including circular wait detection on the complex action named $label_p$ has been performed, the function Depends-On (cf. section 4.5.2.4) can be

used to find dependencies for the action execution call `action` i: *label*`{` $term_{par_1}$, ..., $term_{par_m}$ `}` by calling DEPENDS-ON(`begin(action` i`))`). From the list of triples returned we can extract all dependency time points $timePoint_1$, ..., $timePoint_l$ ignoring duplicates. A time point $timePoint_j$ is either of the form `begin(action` $identifier_{timePoint_j}$`))` or `end(action` $identifier_{timePoint_j}$`))`. With this knowledge we can derive all time points of the former and all time points of the latter kind, i.e. $beginTimePoint_1, \ldots, beginTimePoint_o$, and $endTimePoint_1, \ldots, endTimePoint_p$ respectively. Let $label_{timePoint_j}$ be the label of the action term $term_{timePoint_j}$ associated with `action` $identifier_{timePoint_j}$: $term_{timePoint_j}$ in the associated complex action rule (cf. listing 4.5.1) and let $index_{timePoint_j}$ be the child index of such action execution call. Let $index_i$ be the child index of the action execution call `action` i: *label*`{` $term_{par_1}$, ..., $term_{par_m}$ `}`.

Listing 4.5.5 shows the rather convoluted extension of the basic reactive rule pattern. All dependencies on time points are represented by appropriate queries on events entailed by action execution. As mentioned earlier, the `_composition` term ensures that in fact the correct time points associated with the the correct action execution instances are queried.

Listing 4.5.5: Reactive rule pattern with dependencies

```
ON
  and {
    event identifier_p _begin: label_p$initiated{
      payload{
        id{ parent_ID },
        term_p1 , ..., term_pn
      }
    },
    event identifier_beginTimePoint1 _begin: label_beginTimePoint1$initiated{
      payload{
        _composition{
          child-index{ index_beginTimePoint1 },
          parent-action-id{ parent_ID }
        }
      }
    },
    ⋮
    event identifier_beginTimePointo _begin: label_beginTimePointo$initiated{
      payload{
        _composition{
          child-index{ index_beginTimePointo },
          parent-action-id{ parent_ID }
        }
      }
    },
    event identifier_endTimePoint1 _end: label_endTimePoint1$succeeded{
      payload{
        _composition{
          child-index{ index_endTimePoint1 },
          parent-action-id{ parent_ID }
        }
      }
    },
    ⋮
    event identifier_endTimePointp _end: label_endTimePointp$succeeded{
      payload{
        _composition{
          child-index{ index_endTimePointp },
          parent-action-id{ parent_ID }
        }
      }
    }
  }
DO
  concurrent {
    action: label{
      _composition{
        child-index{ index_i },
        parent-action-id{ parent_ID }
      },
      term_par1 , ..., term_parm
    }
  }
END
```

4.5.3.4. Waiting

As described in section 4.5.2, execution constraints may enforce a certain waiting time between a time point and the execution of an action. This means that the firing of a reactive rule might have to wait until all these waiting times between time points have passed.

Although there is no direct blocking or waiting mechanism in Dura_C we can use negation to achieve the desired effect. Consider the following example: "Execution of action `x` shall wait for the event `e` and at least two minutes after this event." Let `f` be an event of which we know for certain that it will never be generated. Listing 4.5.6 shows how blocking can be expressed.

Listing 4.5.6: Blocking by negation example

```
ON
  and {
    event u: e{},
    not {
      event r: f{}
    } where {
      end(event r) - end(event u) < 2 min
    }
  }
DO
  concurrent {
    action: x{}
  }
END
```

The negation and its its conditions (`where` block) say that no event `f` may be detected the end reception time of which is smaller than the end reception time of a detected `e` plus an additional two minutes.

During runtime, in order to detect whether the event `f` with this property is detected, the runtime system blocks firing of this rule for at least up to close before two minutes after event `e`'s end reception time. Why is that? For example, at 1.8 minutes after `e`'s end reception time the `f` event could possibly be generated. The condition would be satisfied and the negation would cause the entire query to fail. Two minutes after `e`'s end reception time this condition cannot be satisfied anymore thus satisfying the negation and the whole query matches. Now, the runtime system does not know that the event `f` will never be generated, so for the query part of the reactive rule in listing 4.5.6 the runtime system will always wait processing until at least close up to two minutes have passed after event `e`'s end reception time.

Consider again the condition `end(event r) - end(event u) < 2 min`. Its negation is `end(event u) - end(event r) <= -2 min` which is stating that `end(event r)` may occur only after at least two minutes have passed since `end(event u)`. If we replace `end(event r)` with "execution of action `x`" we have exactly the execution constraint stated for the example.

The informally introduced idea of blocking by negation can be used together with our extended reactive rule pattern of listing 4.5.6 and its definitions as described in the following. Let $negationWaitingConditions$ be the conditions for the negation used for waiting. Then listing 4.5.7 shows the final extended reactive rule pattern.

Since we could not assume a standard event to exist that is never generated, such an event is simulated by the extended reactive rule pattern on a per action execution basis. Since we know that an instance of action *label* with given parent ID and child index has not been initiated yet (that is what this reactive rule is for) we can use its entailed initiation event as an event that is never generated up until the rule fires.

The final task of translating the given action execution call is the creation of the conditions $negationWaitingConditions$ for the negation block used for waiting. This is a conjunction of temporal constraints and can be built by using the DEPENDS-ON function and the naming scheme used throughout the earlier sections. The detailed process is described in algorithm 4.5.1. The function BUILD-NEGATIONWAITINGCONDITIONS takes an action execution call identifier and returns the conditions $negationWaitingConditions$ to be used in the pattern of listing 4.5.7.

Algorithm 4.5.1: Building waiting conditions
BUILD-NEGATIONWAITINGCONDITIONS(i)
(1) Initialize $negationWaitingConditions$ as empty
(2) **foreach** $(timePoint, rel, -d) \in$ DEPENDS-ON(`begin`(i))
(3) Initialize rel_{new}
(4) **if** rel is $<$
(5) $rel_{new} \leftarrow$ `<=`
(6) **else**
(7) $rel_{new} \leftarrow$ `<`
(8) **if** $timePoint$ is `begin(action` j`)`
(9) Add `end(event` i`_begin) - end(event` j`_begin)` rel_{new} d to $negationWaitingConditions$
(10) **else if** $timePoint$ is `end(action` j`)`
(11) Add `end(event` i`_begin) - end(event` j`_end)` rel_{new} d to $negationWaitingConditions$
(12) **return** $negationWaitingConditions$

Listing 4.5.7: Reactive rule pattern with dependencies and waiting

```
ON
  and {
    event identifier_p_begin: label_p$initiated{
      payload{
        id{ parent_ID },
        term_p1 , ..., term_pn
      }
    },
    event identifier_beginTimePoint1_begin: label_beginTimePoint1$initiated{
      payload{
        _composition{
          child-index{ index_beginTimePoint1 },
          parent-action-id{ parent_ID }
        }
      }
    },
    ⋮
    event identifier_endTimePoint1_end: label_endTimePoint1$succeeded{
      payload{
        _composition{
          child-index{ index_endTimePoint1 },
          parent-action-id{ parent_ID }
        }
      }
    },
    ⋮
    not {
      event i_begin: label$initiated{
        payload{
          _composition{
            child-index{ index_i },
            parent-action-id{ parent_ID }
          }
        }
      }
    } where {
      negationWaitingConditions
    }
  }
DO
  concurrent {
    action: label{
      _composition{
        child-index{ index_i },
        parent-action-id{ parent_ID }
      },
      term_par1 , ..., term_parm
    }
  }
END
```

4.5.4. Composition Translation

The previous sections discussed the translation of a single action execution call within a complex action to a reactive rule. In order to translate an entire composition of action execution calls the following steps are necessary:

1. Check for sanity of the execution constraints including circular wait detection and build the data structures for the function DEPENDS-ON as described in section 4.5.2.
2. For every action execution call perform the translation steps described in section 4.5.3.
3. Extend the schema of affected action definitions with the schema of the `_composition` subterm.

4.5.5. Success and Failure Specifications

A crucial feature of user-defined actions and complex action rules is the specification of the associated complex action's success or failure. Without a facility to do this, execution of a complex actions would never detectably end, i.e. succeed or fail. This facility is provided in Dura by the `succeeds on` and `fails on` blocks in the supplement $supplement_E$ of an action composition (cf. listing 4.5.1). Without loss of generality, let $suOrFa \in \{\texttt{succeeds on}, \texttt{fails on}\}$, let $eventComposition$ be an event composition, and let $supplementRest_E$ contain the rest of the supplement $supplement_E$ (which contains $suOrFa$). Listing 4.5.8 presents the usage pattern for this construct.

Listing 4.5.8: Success or failure specification pattern

```
FOR
  action identifier_p: label_p{
    term_p1,
    term_p2,
    ⋮
    term_pm
  }
DO
  concurrent {
    action identifier_1: term_1,
    action identifier_2: term_2,
    ⋮
    action identifier_n: term_n
  } suOrFa {
    eventComposition
  } supplementRest_E
END
```

Detection of the complex event defined by $eventComposition$ decides on the failure or success of an execution of this action named $label_p$. Somehow this has to be translated to an event definition (with a deductive rule) called $label_p$`$succeeded` or $label_p$`$failed` respectively.

For reasons to be disclosed later this exact naming shall not be used yet for the basic translation of *eventComposition* in the $suOrFa$ block. Instead, a helper event named $label_h$ is specified. Let $label_h$ be a unique label, let $label_{p_1}$, …, $label_{p_m}$ be the labels of $term_{p_1}$, …, $term_{p_m}$, let $type_{p_1}$, …, $type_{p_m}$ be the parameter types derived from the schema of the action named $label_p$, and let p_{ID}, var_{ch}, var_{par}, var_t, var_{p_1}, …, var_{p_m} be uniquely named variables. Listing 4.5.9 presents the definition of the helper event $label_h$.

The basic idea for this translation is, that in order to derive from the event composition *eventComposition* success or failure of an action instance, an initiation (entailed) event of that action instance needs to be matched and its instance identification propagated. This is why $label_p$`$initiated` and all of its payload is queried as well as $eventComposition$. The complete translation to $label_p$`$succeeded` or $label_p$`$failed` will be discussed further down and will use this helper event named $label_h$.

Listing 4.5.9: Success or failure helper event

```
EVENT
  label_h{
    payload{
      _composition{
        child-index{ integer },
        parent-action-id{ identifier }
      },
      id{ identifier },
      initiation-time{
        begin{ timestamp }
      },
      label_p1{ type_p1 },
      ⋮
      label_pm{ type_pm }
    }
  }
WITH
  DETECT
    label_h{
      payload{
        _composition{
          child-index{ var_ch },
          parent-action-id{ var_par }
        },
        id{ p_ID },
        initiation-time{
          begin{ var_t }
        },
        label_p1{ var_p1 },
        ⋮
        label_pm{ var_pm }
      }
    }
  ON
    and {
      event: label_p$initiated{
        payload{
          _composition{
            child-index{ var_ch },
            parent-action-id{ var_par }
          },
          id{ p_ID },
          initiation-time{
            begin{ var_t }
          },
          label_p1{ var_p1 },
          ⋮
          label_pm{ var_pm }
        }
      },
      event: eventComposition
    }
  END
END
```

4.5.5.1. Handling Action Instance References

The event composition *eventComposition* in the *suOrFa* block is not a standard event composition. It has one distinguishing feature: It allows for references to action execution instances. Let `action` $identifier_i$`:` $term_i$ be an action execution call in the complex action rule of listing 4.5.1. Within the *suOrFa* block one can query for the initiation and success of exactly this action execution instance by querying `action` $identifier_i$`$initiated` and `action` $identifier_i$`$succeeded` respectively. The former can also be done for the initiation of the associated complex action itself by querying `action` $identifier_p$`$initiated`.

Consider the example in listing 4.5.10. It shows how to specify that execution of the complex action `a` shall only succeed if the second execution of action `b` ended successfully.

Listing 4.5.10: Instance reference example

```
FOR
  action i: a{}
DO
  concurrent {
    action j: b{},
    action k: b{}
  } where {
    end(action j) - begin(action k) <= -3 min,
  } succeeds on {
    event: action k$succeeded{}
  }
END
```

Normal event queries are entirely independent of such name scope associated with action instances so these special reference constructs have to be replaced in *eventComposition*. References to instances can be achieved with the instance information contained in the payload (within `_composition` to be exact) of the events entailed by action execution.

Recall the definitions of section 4.5.5. Let p_{ID} be the *same* variable as introduced in that section, let $index_i$ be the child index of `action` $identifier_i$`:` $term_i$, let $label_i$ be the label of term $term_i$. The special query term for the event `action` $identifier_i$`$initiated` might also contain direct subterms $term_{specTop_1}$, ..., $term_{specTop_u}$ as well as subterms of the payload subterm $term_{specPay_1}$, ..., $term_{specPay_v}$, and a query supplement $supplement_Q$. This special query can then be translated as shown in listing 4.5.11. The translation is the same for `action` $identifier_i$`$succeeded` except that $label_i$`$initiated` becomes $label_i$`$succeeded`.

When the execution instance of the parent (complex) action `action` $identifier_p$`:` $term_p$ is referenced, the translation looks different and there are three cases to consider. Let p_{ID} again be the *same* variable as introduced in section 4.5.5, let $label_p$ be the label of term $term_p$. The special query term for the event `action` $identifier_p$`$initiated` might also contain direct subterms $term_{specTop_1}$, ..., $term_{specTop_u}$ as well as subterms of the payload subterm $term_{specPay_1}$, ..., $term_{specPay_v}$. If the payload term in the special query does not contain a subterm labeled `id` the query can be translated as shown in listing 4.5.12.

Listing 4.5.11: Instance reference translation (execution call)

```
event: label_i$initiated{
  payload{
    _composition{
       child-index{ index_i },
       parent-action-id{ p_ID }
    },
    term_specPay1 ,
    ⋮
    term_specPayv
  },
  term_specTop1 ,
  ⋮
  term_specTopu
} supplement_Q
```

Listing 4.5.12: Instance reference translation (parent action) case one and two

```
event: label_p$initiated{
  payload{
    id{ p_ID },
    term_specPay1 ,
    ⋮
    term_specPayv
  },
  term_specTop1 ,
  ⋮
  term_specTopu
} supplement_Q
```

If the payload term in the special query *does* contain a $term_{specPay_i}$ labeled `id` but does not have a variable as child, i.e. merely the subterm's existence is queried, the translation is as shown in listing 4.5.12 but with $term_{specPay_i}$ removed beforehand.

If the payload term in the query *does* contain a $term_{specPay_i}$ labeled `id` and *does* have a variable as child, the translation is slightly more different. Let var_{ID} be the variable contained in $term_{specPay_i}$, i.e. the initiated action's ID, and let $term_{specPay_1}$, ..., $term_{specPay_w}$ be the subterms of the payload subterm *without* $term_{specPay_i}$. Listing 4.5.13 shows the translation for this case.

If any other query in the same block as the special query needs to match with var_{ID} we can both maintain that this is possible as well as the correct matching with the correct action instance by using the `let` block at the end of the event query. As a naive approach one could also just replace all occurrences of var_{ID} with p_{ID} but this might lead to naming conflicts unless renaming only happens within a specific scope. By using a `let` block this problem is solved locally without having to worry about renaming issues.

Listing 4.5.13: Instance reference translation (parent action) case three

```
event: label_p$initiated{
  payload{
    id{ p_ID },
    term_specPay_1 ,
    ⋮
    term_specPay_w
  },
  term_specTop_1 ,
  ⋮
  term_specTop_u
} let {
  var identifier var_ID = p_ID
} supplement_Q
```

Finally, we need to check whether the composition an instance reference is part of has `group by` blocks in its supplement. Let e be the identifier of the special instance query in such a case. To every `group by` block that contains `event` e we need to add p_{ID}. This makes sure that if the special query is properly exposed, i.e. grouped by, the required variable p_{ID} is also exposed. A programmer cannot do this manually since the variable p_{ID} is unknown before applying the transformation described in this section.

The translations described here will during runtime have the desired outcome due to the fact that all translated references share the same variable p_{ID} which is matched in a query to the initiation event of the complex action on a higher level in the helper event named $label_h$. Additionally, the child index information that can be inserted as constants maintains that the correct action execution instances within the action composition are referenced.

Note that due to the way the action instance references are translated, the way nested event compositions are translated (cf. section 4.7), and the fact that we could not find an alternative, there are restrictions on their usage: Whenever one of them is used somewhere in an element of a (possibly nested) disjunction within *eventComposition* every other element in that disjunction must contain an action instance reference (not necessarily the same). Also, groupings and negations may not hide a special instance query to upper layers of composition. These restrictions could be lifted if Dura_C offered nested event compositions without the restrictions our implementation (cf. section 4.7) currently imposes.

4.5.5.2. Ensuring Single Occurrence

An entailed event stating success or failure should only be generated at most once per action execution instance. Querying *eventComposition* in the *suOrFa* block might cause several matchings and, if directly translated, might generate several success or failure events during runtime. One could take the stance that it is a programmer's responsibility to somehow maintain that this does not happen. Instead, we decided to create a translation with the following semantics: The earliest event created by a successful query of *eventComposition* and only this shall cause the deduction of a success or failure event.

This is were the helper event named $label_h$ comes into play. Consider this event already defined (as per translation described earlier) and the action instance references in *eventComposition* resolved (also as described earlier). This helper event named $label_h$ is the raw form of the success or failure event, i.e. one that could possible create several successes or failures for the exact same action instance.

Recall the helper event definition in listing 4.5.9 and the names and definitions used therein. If $suOrFa$ is `succeeds on`, let *outcome* be `succeeded`, otherwise `failed`. Then the event $label_p\$outcome$ can be defined as shown in listing 4.5.14. The schema definition has been omitted here for the sake of brevity. Except for the label it is the same schema as the one for the helper event.

If a query for the helper event containing the ID of an action instance returns an instance (`event a`) it is checked whether there exists no other instance of the helper event (`event b`) with the same action instance ID that ended before. This means that only for that first instance the entire query is satisfied and only for this first instance an instance of $label_p\$outcome$ is derived.

If the other event (`event b`) ended (occurred) at the same time, according to what their reception time values state, the runtime system still must have created one instance entry before the other, which is represented by the system-given ID which has to be awarded to entries in ever increasing monotonous values for this translation to work as intended. Only for the first of these two instances ending at the same time the condition is true that there is no other with a lower ID.

This solution comes with a problem not to be neglected. Garbage collection will most likely not work correctly, or will at least have a hard time with this since the runtime system might never know for certain whether it will never have to perform these checks on past instances again. Alternative solutions with sateful objects have been considered but they come with their own set of issues due to the possible delay between real times and modeled event times. Possibly Dura_C could be extended with an additional facility specifying the behavior we painstakingly expressed by the translation in listing 4.5.14.

Yet another important aspect to consider is the fact that due to the way we translate the failure and success specification, an entailed success or failure event of a user defined (complex) action will always have a begin reception time that is the same as or lower (before) than the begin reception time of the entailed initiation event. This is because we are querying for the initiation event to match the correct action instance. However, the actual success time can be retrieved from the end reception time. Ideally the reception times of entailed success or failure events should be time points, i.e. their begin reception time and end reception time are equal. However, this cannot be achieved with the current limitations of Dura_C. Even if one could avoid querying for the initiation event, the query in the success or failure specification cannot be guaranteed to result in an event instance with a time point reception time. It is even possible that the begin reception time of that instance is lower (before) than the initiation event instance's begin reception time. Again, we find it unlikely that this can be solved without extending Dura_C with special constructs. The question remains whether it needs to be solved at all or whether this behavior is in fact desired. However, answering this question in-depth as well as proposing concrete extensions to Dura_C goes beyond the scope of this chapter.

The concepts described above for handling success and failure specification also apply to the translation of `succeeds on` and `fails on` blocks in action definitions for external actions. Their translation is not further elaborated on in any transformation section of its own in this thesis.

Listing 4.5.14: Ensuring single occurrence translation

```
EVENT
  label_p$outcome{ ... }
WITH
  DETECT
    label_p$outcome{
      payload{
        _composition{
          child-index{ var U },
          parent-action-id{ var V }
        },
        id{ p_ID },
        initiation-time{
          begin{ var W }
        },
        label_p1{ var_p1 },
        ⋮
        label_pm{ var_pm }
      }
    }
  ON
    and {
      event a: label_h{
        id{ var A },
        payload{
          _composition{
            child-index{ var U },
            parent-action-id{ var V }
          },
          id{ p_ID },
          initiation-time{
            begin{ var W }
          },
          label_p1{ var_p1 },
          ⋮
          label_pm{ var_pm }
        }
      },
      not {
        event b: label_h{
          id{ var B },
          payload{
            id{ p_ID }
          }
        }
      } where {
        end(event b) <= end(event a),
        var B < var A
      }
    }
  END
END
```

4.5.6. Complete Transformation

The previous sections describe how single complex action rules and all their parts are translated. Similar to the other transformations described in this chapter, these single translations need to be performed on all complex action rules in a program's AST. The following steps outline the complete transformation:

1. Collect all complex action rules.
2. For every collected complex action rule translate it as described in section 4.5.3.
3. Remove all complex action rules.

4.6. Simplifying Reactive Rules

After applying the transformations described in the previous sections, only reactive rules with non-nested concurrent execution blocks are found in the AST. However, the flat action execution calls in the head of these rules may contain parameters with arithmetic expressions, which Dura_C does not accept. The transformation described in this section resolves this issue by simplifying reactive rules in such a manner that the problem of arithmetic expressions in the head of reactive rules is reduced to solving the problem of arithmetic expressions in the head of deductive rules which will be resolved in yet another, later transformation.

4.6.1. Flat Execution Calls Reactive Rule

Let *eventComposition* be an event composition and $term_1$, ..., $term_n$ be action terms of user-defined, inbuilt internal, or external actions. Using these definitions, listing 4.6.1 shows the general reactive rule pattern with flat action execution calls.

Listing 4.6.1: Flat execution calls reactive rule pattern

```
ON
  eventComposition
DO
  concurrent {
    action: term1,
    ⋮
    action: termn
  }
END
```

The terms $term_1$, ..., $term_n$ might contain parameters that are arithmetic expressions or identifiers from the query part of the reactive rule (such as `event x`). While the latter is accepted by Dura_C the former is not.

One might notice that the execution calls do not have identifiers anymore. They are assumed to have been removed in an intermediate transformation before. Since this transformation is trivial it was not given a section of its own. They are removed because Dura_C ignores all identifiers in the concurrent execution block of reactive rules anyway. Action execution instances cannot be related to each other in Dura_C's reactive rules.

4.6.2. Simplification

Consider the example presented in listing 4.6.2. A quite possible and correct translation could use a `let` block in the supplement of the event composition and produce the code in listing 4.6.3.

We already know that a similar problem, that of expressions in deductive rule heads needs to be resolved, too. Since we do this in a later transformation (cf. section 4.10) we can in

Listing 4.6.2: Expression in parameter example

```
ON
  and {
    event: q{ r{ var A } }
  }
DO
  concurrent {
    action: x{ p{ var A } },
    action: y{ q{ var A * 2 } }
  }
END
```

Listing 4.6.3: Expression in parameter example solution

```
ON
  and {
    event: q{ r{ var A } }
  } let { var _2_0 = var A * 2 }
DO
  concurrent {
    action: x{ p{ var A } },
    action: y{ q{ var _2_0 } }
  }
END
```

the current transformation described in this section change reactive rules so that they are accepted by Dura$_C$. This is done by creating a new event definition with a deductive rule the head of which contains the respective expressions, querying this event in the query part of the reactive rule, and replacing all expressions in the execution part with corresponding variables. To this end we can reuse the function Build-VarToPar (cf. algorithm 4.4.1) introduced in section 4.4.1.2.

Let *varToPar* be the mapping between variables and parameters built with BuildVar-ToPar from the action composition in the execution part of the reactive rule presented in listing 4.6.1. Let *label* be a new unique label name, let $label_{v_1}, \ldots, label_{v_k}$ be unique label names, and let $v_1, \ldots, v_k$ be all the variables of the rule body accessible in the domain or key set of *varToPar*, and let $type_{v_1}, \ldots, type_{v_k}$ be the types of these variables. For a variable v_i let p_i be the corresponding parameter found in *varToPar*, i.e. $(v_i, p_i) \in varToPar$. A new event definition for the event named *label* as presented in listing 4.6.4 is added to the AST.

The original reactive rule (cf. 4.6.1) is simplified as presented in listing 4.6.5. Recall, that in the process of building the *varToPar* mapping, all expressions and identifiers within execution parameters in the terms $term_1$, …, $term_n$ have been replaced by variables v_1, …, v_k.

Listing 4.6.4: Event definition for *queryComposition*

```
EVENT
  label{
    label_v1{ type_v1 },
    ⋮
    label_vk{ type_vk }
  }
WITH
  DETECT
    label{
      label_v1{ p_1 },
      ⋮
      label_vk{ p_k }
    }
  ON
    eventComposition
  END
END
```

Listing 4.6.5: Simplified flat execution reactive rule

```
ON
  and {
    event: label{
      label_v1{ v_1 },
      ⋮
      label_vk{ v_k }
    }
  }
DO
  concurrent {
    action: term_1,
    ⋮
    action: term_n
  }
END
```

4.6.3. Complete Transformation

The complete translation is very simple and can be described with the following steps:

1. Collect all reactive rules.
2. For every collected reactive rule create a new event definition and simplify the reactive rule as described in section 4.6.2.

4.7. Resolving Nested Event Compositions

In order to simplify the construction of complex events, Dura allows for nesting event compositions. Nested event compositions are to be used in settings where querying a complex event is desired but writing a new event definition with schema and deductive rule would be too cumbersome. This is indicated in those cases where an event compositions is used in a single instance only.

Since Dura$_C$ does not support nested event compositions, certain limitations are imposed on their usage regarding negation as well as disjunctive composition. The translation of Dura's nested event compositions to flat queries and the mentioned limitations are discussed in detail in the following.

4.7.1. Nested Event Composition in Positive Queries

Let $op \in \{\texttt{and}, \texttt{or}\}$ be an event composition (conjunction or disjunction) operator, let $Q = \{query_1, \ldots, query_m\}$ be a set of event or state queries in a query composition (conjunction or disjunction), in regard to an arbitrary query $query_i \in Q$ let $E_Q(query_i) = Q \setminus query_i$ be its so-called environment, and let $supplement_Q$ be the supplement to the composition encompassing Q.

Listing 4.7.1 shows the general event composition pattern using the definitions introduced above.

Listing 4.7.1: Event composition pattern

```
op {
  query_1 ,
  ⋮
  query_i ,
  ⋮
  query_m
} supplement_Q
```

The queries in Q may be positive or negative, atomic (i.e. flat) or themselves contain event compositions. In each resolution step to be described in this section only a single nested event composition shall be resolved. As such, for now it suffices to consider such composition without regard to the exact nature of the other queries in its environment.

Listing 4.7.2: Query with nested event composition `event` e

```
event e: op_e {
  query_e1 ,
  ⋮
  query_en
} supplement_Qe
```

Let $op_e \in \{\texttt{and}, \texttt{or}\}$ be an event composition operator, let $Q_e = \{query_{e_1}, \ldots, query_{e_n}\}$ be a set of event or state queries in a composition and let $supplement_{Q_e}$ be the supplement to the composition encompassing Q_e. Then let $query_i$ be the query shown in listing 4.7.2.

Listing 4.7.3: Query in outer event composition

```
op {
  query_1 ,
  ⋮
  event e: op_e {
    query_e1 ,
    ⋮
    query_en
  } supplement_Qe ,
  ⋮
  query_m
} supplement_Q
```

Listing 4.7.3 shows the expanded query $query_i$ within the outer event composition described earlier, encompassing Q. Bindings, i.e. variables and identifiers, outside and inside a nested event composition are matched as usual via same naming. However, bindings occurring in the outer composition cannot be referenced from conditional supplements of the nested event composition.

The general idea for resolving this situation, i.e. removing nested event compositions by transformation into flat queries, is to first name the event composition and turn it into a new complex event by means of a new event definition with a deductive rule. Subsequently, the original query $query_i$, containing the nested event composition, is substituted with an atomic (by name) query referring to the complex event defined by the newly created event definition and deductive rule.

Let $label$ be a new unique label name, let $label_{b_1}, \ldots, label_{b_k}$ be unique label names, and let $b_1, \ldots, b_k$ be all the bindings (state identifiers, event identifiers, variables) of the nested event composition in $query_i$ visible (or accessible) at the end of the nested event composition after its supplement. Let $type_{b_1}, \ldots, type_{b_k}$ be the types of these bindings which can be inferred by the usage of these bindings within the rule body. Listing 4.7.4 shows the formerly nested event composition of $query_i$ in a new complex event definition with deductive rule. Finally, substitution of the nested event composition of $query_i$ in the original, outer composition yields listing 4.7.5.

Listing 4.7.4: Event composition of $query_i$ as named complex event

```
EVENT
  label{
    label_b1{ type_b1 },
    ⋮
    label_bk{ type_bk }
  }
WITH
  DETECT
    label{
      label_b1{ b1 },
      ⋮
      label_bk{ bk }
    }
  ON
    op_e {
      query_e1,
      ⋮
      query_en
    } supplement_Qe
  END
END
```

Listing 4.7.5: Event composition in $query_i$ substituted

```
op {
  query1,
  ⋮
  event e: label{
    label_b1{ b1 },
    ⋮
    label_bk{ bk }
  },
  ⋮
  query_m
} supplement_Q
```

4.7.1.1. Redundant Bindings

One apparent shortcoming of the resolution idea described above is the creation of an unnecessarily large amount of bindings and terms. Not all of the variables occurring within a nested event composition are relevant for matching with bindings of its outer environment.

Consider the example shown in listing 4.7.6. Omitting the actual definition of the complex event and deductive rule for the sake of brevity, the naive resolution would yield a substitution resulting in something similar (save for the actual label naming) to listing 4.7.7.

Listing 4.7.6: Redundant bindings example

```
and {
  event a: u{ p{ var A } },
  event b: and {
    event f: v{ p{ var A } },
    event g: w{ p{ var B } },
    event h: x{ p{ var C } }
  }
}
```

Listing 4.7.7: Redundant bindings example

```
and {
  event a: u{ p{ var A } },
  event b: _3_0{
    _3_0_0{ event f },
    _3_0_1{ event g },
    _3_0_2{ event h },
    _3_0_3{ var A },
    _3_0_4{ var B },
    _3_0_5{ var C }
  }
}
```

Here it would suffice for the event named `_3_0` to merely expose those bindings that are relevant for matching in its environment as shown in listing 4.7.8.

Listing 4.7.8: Relevant bindings example

```
and {
  event a: u{ p{ var A } },
  event b: _3_0{ _3_0_0{ var A } }
}
```

This might seem like an early optimization that should rather be delegated to a different transformation or entirely to the compilation of Dura_C. After all, a binding usage analysis could be employed to identify unused bindings in general and adapt all rule heads and atomic queries accordingly. However, three advantages of taking this aspect into account within this transformation step itself can be identified:

- Due to full control over the creation of a unique named event composition a complicated analysis across all possible matchings in all compositions is not necessary.
- The size of the resulting abstract syntax tree can be reduced.
- An analysis of binding usage in regard to the immediate (local) environment can be useful for later transformations.

Finding relevant bindings first requires that we find all bindings in a query. The implementations of the functions needed in the algorithm for collecting these bindings are fairly

trivial. Most of them simply return immediate children of a subtree of the abstract syntax tree provided:

- Identifier shall return identifier associated with a query.
- Term shall return the query term of an atomic query or the head term of a deducitve rule head.
- Term-Bindings shall return the bindings contained in a term.
- Composition shall return the composition (`and` or `or` node) of a query with nested event composition.
- Queries shall return the subqueries contained in a composition.
- Supplement shall return a composition's, query's, or head's supplement.
- Supp-Defs shall return the variable definitions contained in a supplement (occurring in let or aggregate statements).

Assuming the existence of these basic syntactic decomposition functions, algorithm 4.7.1 collects all positive i.e. not negated or existentially quantified) bindings within a positive or negative query that can be matched by other queries in its environment. The question on why binding collection does not follow further nested event compositions within negative queries, as well as why conjunctions and disjunctions are handled differently will be revisited later.

Algorithm 4.7.1: Collecting bindings.
Bindings(*query*)
(1) **if** *query* is negative
(2) *query* ← the positive query within the negative query *query*
(3) $Q \leftarrow \{$ Identifier(*query*) $\}$
(4) **if** *query* is atomic
(5) $Q \leftarrow Q \cup$ Term-Bindings(Term(*query*))
(6) **return** $Q \cup$ Supp-Defs(Supplement(*query*))
(7) **else**
(8) **if** Composition(*query*) is a conjunction
(9) **foreach** $q \in$ Queries(Composition(*query*))
(10) **if** q is positive
(11) $Q \leftarrow Q \cup$ Bindings(q)
(12) **else**
(13) $R \leftarrow \emptyset$
(14) **foreach** $q \in$ Queries(Composition(*query*))
(15) **if** q is positive
(16) **if** R is empty
(17) $R \leftarrow$ Bindings(q)
(18) **else**
(19) $R \leftarrow R \cap$ Bindings(q)
(20) $Q \leftarrow Q \cup R$
(21) **return** $Q \cup$ Supp-Defs(Supplement(Composition(*query*)))

Collecting the bindings of positive queries would suffice to implement the naive resolution of nested event compositions. In order to purge such a collection of redundant bindings for a query with nested event composition, the query's environment needs to be considered. Algorithm 4.7.2 does just this. The result of RELEVANT-BINDINGS is the set of bindings to be exposed in the head term of the deductive rule in a new event definition, used for resolving the nested event composition. If we calculate the relevant bindings of a positive query p we need to consider *all* other queries in its environment, even negative ones. This is because such a negative query might need to match with bindings of the positive query q. However, if we calculate the relevant bindings of a negative query n we only need to consider other *positive* queries in its environment. This is because bindings in n can only match with positive bindings in its environment. Furthermore n itself cannot expose new bindings not found in other positive queries in its environment.

Algorithm 4.7.2: Filtering for relevant bindings.
RELEVANT-BINDINGS(*query*, *head*, *Environment*, *supplement*)
(1) $B \leftarrow$ BINDINGS(*query*)
(2) $Q \leftarrow$ TERM-BINDINGS(TERM(*head*))
(3) **if** *query* is positive
(4) $Q \leftarrow Q \cup$ SUPP-REFS(SUPPLEMENT(*head*))
(5) $Q \leftarrow Q \cup$ SUPP-REFS(*supplement*)
(6) **foreach** $q \in$ *Environment*
(7) **if** *query* is positive **or** q is positive
(8) $Q \leftarrow Q \cup$ BINDINGS(q)
(9) **return** $B \cap Q$

Relevance of a binding is determined not only by the environment queries but also by the binding references occurring in the outer composition supplement and not to forget the bindings occurring in the head of the deductive rule the nested event composition is found in. Hence, for a given query *query* the following new functions and parameters are required:

- SUPP-REFS shall be defined as returning all the bindings referenced (not defined) in a supplement.
- *Environment* shall be a *query*'s environment as defined at the beginning of this section. On the level of the abstract syntax tree the environment is merely the collection of all the siblings of a query node inside a conjunction or disjunction.
- *supplement* shall be the supplement to the composition that *query* is contained in if *query* is positive, which is assumed for now. The negative case will be revisited in a later section.
- *head* shall be the head of the deductive rule the body of which contains the composition that in turn contains *query*. Note that *head* is not defined in relation to the composition containing *query* but to the whole deductive rule. Note that the head may contain a supplement, i.e. grouping.

Again, details for the implementation of the function SUPP-REFS are not given here since it is considered trivial. It is merely a collection of all bindings. In `let` blocks only the right hand sides of variable definitions are searched.

Listing 4.7.9: Relevant bindings example

```
DETECT
  // head
  t{ p{ var B } }
ON
  and {
    // query1
    event a: u{ p{ var X } },
    // query2
    event b: and {
      event f: v{ p{ var X } },
      event g: w{ p{ var Y } }
      event h: x{ p{ var Z } }
    },
    // query3
    event c: or {
      event i: y{},
      event j: z{ p{ var W } }
    },
    // query4
    event d: and {
      event k: a{ p{ var A } },
      event l: b{ p{ var B } }
    }
  } // supplement
    where { var A > var X }
END
```

For illustration purposes consider the example presented in listing 4.7.9. In order to resolve the three complex subqueries $query_2$, $query_3$ and $query_4$, relevant bindings need to be calculated for each of them as follows:

$$\text{Relevant-Bindings}(query_2, head, \{query_1, query_3, query_4\}, supplement) = \{\texttt{var X}, \texttt{var Y}\}$$

$$\text{Relevant-Bindings}(query_3, head, \{query_1, query_2, query_4\}, supplement) = \emptyset$$

$$\text{Relevant-Bindings}(query_4, head, \{query_1, query_2, query_3\}, supplement) = \{\texttt{var A, var B}\}$$

Since $query_1$ is atomic its relevant bindings do not need to be calculated and although calling Relevant-Bindings on it is possible, it has no meaning for the transformation described in this section.

Note that in the actual prototype implementation extensive usage of saving intermediate results such as those of collecting all bindings is employed in order to avoid redundant recalculations.

4.7.1.2. Resolution

The functions and methods described in section 4.7.1.1 allow for improved resolution of a single nested event composition in a query *query* within the body of a deductive rule *rule* by the following steps:

1. Calculate the relevant bindings for *query*.
2. Create a new deductive rule $rule_{new}$ with the composition and composition supplement of *query* as the body.
3. Give $rule_{new}$ a unique head term label and for each relevant binding add a new subterm to the head term containing the binding.
4. Add a new complex event definition for $rule_{new}$ to the AST.
5. In the original composition in the body of *rule* containing *query* substitute the event composition in *query* with the head term of $rule_{new}$.

4.7.1.3. Negative Subqueries and Bindings

The resolution process as described above in section 4.7.1.2 has an important implication on negative queries within nested event compositions. Consider the example shown in listing 4.7.10.

Listing 4.7.10: Negative subquery example (valid)

```
and {
  event a: u{ p{ var X } },

  // query
  event b: and {
    event f: v{
      w { var X },
      x { var Y }
    },
    not {
      event g: y{ p{ var X } }
    } where { event g during event f }
  }
}
```

The (positive) bindings in *query* (`event b: ...`) are `event f`, `var X`, and `var Y`. So, the only relevant binding is `var X`. Since `var X` occurs both positively and negatively within the nested event composition it is no problem to expose it in the head of the deductive rule to be created.

Now suppose `var X` only occurs negatively as shown in listing 4.7.11. The (positive) bindings in *query* (`event b: ...`) are `event f` and `var Y`. Since `var X` only occurs negatively there

Listing 4.7.11: Negative subquery example (problematic)*

```
and {
  event a: u{ p{ var X } },

  // query
  event b: and {
      event f: v{ p{ var Y } },
    not {
      event g: w{ p{ var X } }
    } where { { event f, event g } within 2 min }
  }
} where { event b after event a }
```

are no relevant bindings for *query*. This behavior is intentional and the reason for this is described in the following[2].

In Dura, bindings of same name are matched and one might want to offer this feature for bindings solely occuring within a negated subquery. However, due to limitations, such as range restriction, imposed by TSA (and in turn Dura_{C}) such situation cannot be resolved. One is tempted to resolve it by inserting a copy of the query's environment to the nested composition as presented in listing 4.7.12.

Listing 4.7.12: Inserted environment example

```
and {
  event a: u{ p{ var X } },

  // query
  event b: and {
      // inserted environment
      event a: u{ p{ var X } },

      event f: v{ p{ var Y } },
    not {
      event g: w{ p{ var X } }
    } where { { event f, event g } within 2 min }
  }
} where { event b after event a }
```

At first glance this seems to resolve the issue regarding the matching of bindings. Now the nested event composition in *query* could be resolved as described earlier. However, a new problem is introduced. The temporal condition `event b after event a` in the example above cannot be satisfied. This is due to the implicit reception times of the conjunctive nested event composition. Its begin and end times always encompass the time of `event a`, so `event b` is never detected after `event a`. The semantic would change entirely.

[2]In fact it will be concluded that a program like listing 4.7.11 shall not be considered a valid Dura program. All such (invalid program) listings are marked with a * sign

There are of course cases where this is not an issue, for instance if there is no such temporal constraint. However, a general solution would call for circumventing the implicit reception time calculation, manually constraining it to non-environment events, returning it as variables and adapting the outer temporal constraints to those manually calculated times.

Listing 4.7.13: Interdependent subqueries example*

```
and {
  // query1
  event a: and {
      event f: v{ p{ var X } },
    not {
      event g: w{ p{ var X } }
    } where { event g during event f }
  },

  // query2
  event b: and {
      event h: x{ p{ var Y } },
    not {
      event i: y{ p{ var X } }
    } where { event i during event h }
  } where { event i during event h }
}
```

A further complication with this idea is illustrated in listing 4.7.13. If negative bindings are to be matched with positive ones of same name, the complex subqueries $query_1$ and $query_2$ are interdependent. Stepwise resolution of complex subqueries might solve such situations, but especially in large and deeply nested event compositions this resolution is likely to create an intractable amount of new deductive rules.

Most importantly, as was indicated in the introduction to this section, for all intents and purposes nested event compositions are to be seen as mere placeholders for (named) complex events (further specified by deductive rules). Since these simplified semantics of nested event compositions shall be retained, the issue of bindings of same name that occur negatively within and positively outside the nested composition can be addressed by either of the following solutions:

- Impose the following limitation: A negative binding within a nested event composition is allowed only if the same binding occurs positively within the same nested event composition or if it does not occur anywhere else in the whole deductive rule except for the associated negative subquery's composition supplement.
- Ignoring the fact that a matching binding exists in the environment or the supplements, consider a negative binding that only occurs negatively within a nested event composition as independent, i.e. as if it had a unique name.

As the latter is contrary to the general intuition employed throughout Dura and Dura_C, that bindings are matched according to same naming, the former solution or limitation is imposed on Dura programs.

4.7.1.4. Grouping in Nested Event Compositions

Grouping often leads to a reduction of exposed bindings after the grouping operation has been applied, due to the nature of limiting and rearranging the result sets. Consider the example program shown in listing 4.7.14. Resolving the nested composition as described in section 4.7.1.2 results in a new event definition and a substitution in the original code (cf. listing 4.7.15).

Listing 4.7.14: Nested event composition with grouping example

```
DETECT
  a{}
ON
  and {
    event a: u{ p{ var X } },
    event b: and {
      event f: v{ p{ var Y } },
      event g: w{ p{ var X } }
    } where { { event f, event g } within 2 min }
      group by { event f }
  }
END
```

Listing 4.7.15: Resolved subquery (substitution & rule)*

```
DETECT
  a{}
ON
  and {
    event a: u{ p{ var X } },
    event b: _3_0{ _3_0_0{ var X } }
  }
END

EVENT
  ...
WITH
  DETECT
    _3_0{ _3_0_0{ var X } }
  ON
    and {
      event f: v{ p{ var Y } },
      event g: w{ p{ var X } }
    } where { { event f, event g } within 2 min }
      group by { event f }
  END
END
```

At the end of the conjunction and after grouping, the only visible or accessible binding is `event f` due to grouping by `event f`. However `var X` is referred to in the head of the rule.

As is, this violates Dura_C's range restriction requirement and is thus not valid in Dura or Dura_C.

There is no doubt that this piece of code does something the programmer might not have considered when writing the original nested event composition. So, however this situation is handled, in any case the programmer has to be informed about the issue. There are two possible solutions:

- Throw an error (during semantic analysis) with a detailed message and abort compilation.
- Recover the lost (or hidden) bindings according to what a programmer most likely intended and issue a detailed warning.

The former would surely be the easiest and bluntest way to handle the situation. In fact, this would not have to be handled at all by the translator from Dura to Dura_C since it is bound to fail error analysis. As to the latter option, if programmers are kept aware of the semantic adaptations occurring during recovery of hidden bindings (cf. listing 4.7.16), allowing them to omit explicit mentioning of bindings (to be exposed) in the `group by` construct, may help keep the program code concise.

Listing 4.7.16: Recovered bindings

```
EVENT

WITH
  DETECT
    _3_0{
      _3_0_0{ var X }
    }
  ON
    and {
      event f: v{ p{ var Y } },
      event g: w{ p{ var X } }
    } where { { event f, event g } within 2 min }
      group by { event f, var X }
  END
END
```

This recovery of hidden bindings has been experimentally implemented in our prototype and this option is open for consideration. However, for the time being it is not to be employed.

4.7.1.5. Nested Disjunctive Event Compositions

The resolution presented in section 4.7.1.2 already handles queries containing a (nested) disjunctive event composition (i.e. $op_e = \texttt{or}$), but yet again there are certain implications to be considered.

Due to how bindings are collected from such disjunctions (by set intersection instead of union) only those bindings that occur within every subquery will be matchable outside the nested composition and considered for calculating relevant bindings. Consider the example

presented in listing 4.7.17. Since `var X` occurs in all subqueries it can be exposed and a resolution might yield the substitution shown in listing 4.7.18.

Listing 4.7.17: Nested disjunctive event compositions example (valid)

```
and {
  event a: u{ p{ var X } },
  event b: or {
    event f: v{
      w { var X },
      x { var Y }
    },
    event g: w{ p{ var X } }
  }
}
```

Listing 4.7.18: Resolved disjunctive event composition example

```
and {
  event a: u{ p{ var X } },
  event b: _3_0{
    _3_0_0{ var X }
  }
}
```

However, if a binding occurs outside (e.g. `var X` in `event a: u{ var X }` and in only one of the subqueries one runs into issues with binding matchability similar to the ones described in section 4.7.1.3. Listing 4.7.19 shows such a problematic case[3]. The substituted query will not expose or contain any bindings although the general intuition of Dura and Dura_C would suggest that the outer variable is matched with the inner.

Listing 4.7.19: Nested disjunctive event compositions example (problematic)*

```
and {
  event a: u{ p{ var X } },
  event b: or {
    event f: v{ p{ var Y } },
    event g: w{ p{ var X } }
  }
}
```

Again, due to the range restriction limitation of TSA and Dura_C the described situation cannot be solved elegantly. One could try to solve it by transforming the code into the equivalent of the disjunctive normal form (DNF) in Dura syntax (cf. listing 4.7.20). However, one can easily create a situation that will result in an intractably large composition size, in effect causing the translation from Dura to Dura_C to take virtually forever, that

[3] In fact it will be concluded that a program like listing 4.7.19 shall not be considered a valid Dura program. All such (invalid program) listings are marked with a * sign

Listing 4.7.20: DNF solution example

```
or {
  event p: and {
    event a_0: u{ p{ var X } },
    event   f: v{ p{ var Y } }
  },
  event p: and {
    event a_1: u{ p{ var X } },
    event   g: w{ p{ var X } }
  }
}
```

is if it does not abort due to running out of memory earlier. Let $query_{e_{1_a}}$, ..., $query_{e_{n_a}}$ be queries that do not expose the variable `var X`. Listing 4.7.21 presents a general pattern from which the described intractability can arise. Translating this pattern into DNF will yield a disjunction containing 2^n queries with nested conjunctive event composition.

Listing 4.7.21: Intractable disjunctive complex subqueries

```
and {
  event e_1: or {
    query_{e_{1_a}} ,
    event e_{1_b}: label_{e_1}{ p{ var X } }
  },
  ⋮
  event e_n: or {
    query_{e_{n_a}} ,
    event e_{n_b}: label_{e_n}{ p{ var X } }
  }
}
```

Since intractability is not desirable, we decided against translating aforementioned situations to the DNF. There are two possible options to address the issue:

- Impose the following limitation: A binding within a subquery of a nested disjunctive event composition is allowed to be used only if the same binding is exposed by all subqueries within the same composition or if it does not occur anywhere else in the whole deductive rule except for the subquery's supplement.
- Ignoring the fact that a matching binding exists in a query's environment or the supplements, consider a binding that does not occur in all subqueries within a nested disjunctive event composition as independent, i.e. as if it had a unique name.

As the latter is contrary to the general intuition employed throughout Dura and Dura$_\mathrm{C}$, that bindings are matched according to same naming, the former solution is pursued and the described limitation is imposed on Dura programs.

4.7.2. Nested Event Composition in Negated Queries

While section 4.7.1 dealt with nested event compositions within either a disjunction (`or`) or conjunction (`and`), this section only deals with nested event compositions (as part of negated queries) within conjunctive compositions. This is due to the fact that negation requires temporal constraints relating the negated query with a positive query of its environment. In disjunctions this is meaningless since only one of the queries within a disjunction needs to match and therefore no relationship can be established between two queries within a disjunction.

Let $Q = \{query_1, \ldots, query_n\}$ be a set of event or state queries in a conjunction, let $query_i \in Q$ be a negated query, in regard to $query_i$ let $E_Q(query_i) = Q \setminus query_i$ be its so-called environment, let *head* be the head term of the deductive rule the body of which contains the conjunction, and let $supplement_Q$ be the supplement to the conjunction encompassing Q. Listing 4.7.22 shows the general conjunctive event composition pattern with negated subqueries using the definitions introduced above.

Listing 4.7.22: Disjunctive composition pattern

```
DETECT
  head
ON
  and {
    query_1 ,
    ⋮
    query_i ,
    ⋮
    query_n
  } supplement_Q
END
```

Let *eventComposition* be an event composition and let $supplement_{neg}$ be a $query_i$'s negation supplement. Let $query_i$ be the negated query shown in listing 4.7.23. Listing 4.7.24 shows the expanded query $query_i$ within the outer composition described earlier, encompassing Q.

Listing 4.7.23: Negated query `event` e

```
not {
  event e: eventComposition
} supplement_neg
```

Listing 4.7.24: Nested event composition in `event` e

```
DETECT
  head
ON
  and {
    query1 ,
    ⋮
    not {
      event e: eventComposition
    } supplement_neg ,
    ⋮
    query_n
  } supplement_Q
END
```

4.7.2.1. Resolution

Due to the way the functions for calculating bindings and relevant bindings are described in section 4.7.1.1, only a small adaption is required for handling nested event compositions in negated queries.

Let *label* be a new unique label name, let $label_{b_1}, \ldots, label_{b_k}$ be unique label names, let $supplement_{neg}$ be the negation's supplement, and let RELEVANT-BINDINGS($query_i$, *head*, $E_Q(query_i)$, $supplement_{neg}$) $= \{b_1, \ldots, b_k\}$ be the set of relevant bindings (state identifiers, event identifiers, variables) (cf. algorithm 4.7.2) for the negated query $query_i$ containing *eventComposition*. Let $type_{b_1}, \ldots, type_{b_k}$ be the types of these bindings which can be inferred by the usage of the bindings within the associated rule's body.

Listing 4.7.25: Nested event composition *eventComposition* as named complex event

```
EVENT
  label{
    label_b1{ type_b1 },
    ⋮
    label_bk{ type_bk }
  }
WITH
  DETECT
    label{
      label_b1{ b1 },
      ⋮
      label_bk{ bk }
    }
  ON
    eventComposition
  END
END
```

Listing 4.7.25 shows the nested event composition of $query_i$ in a new (uniquely named) event definition with a deductive rule. Finally, substitution of *eventComposition* in the original composition yields listing 4.7.26.

Listing 4.7.26: Nested event composition in $query_i$ substituted

```
DETECT
  head
ON
  and {
    query_1 ,
    ⋮
    not {
      event e: label{
        label_b1{ b1 },
        ⋮
        label_bk{ bk }
      }
    } supplement_neg ,
    ⋮
    query_n
  } supplement_Q
END
```

The only real adaption compared to handling a nested event composition as part of a positive query is the non-consideration of bindings in any supplements except for the negation's supplement within the calculation of the required bindings, i.e. the last parameter for RELEVANT-BINDINGS is the negation's supplement. While bindings within the negated query have to be matchable to positive bindings outside, they can never be constrained in (or from) the outer supplement $supplement_Q$ or the head's supplement, but only by $supplement_{neg}$. Note that in the proposed resolution the state or event identifier (e) remains unaltered.

The steps for resolving a nested event composition in a negated query *query* are as follows:

1. Calculate the relevant bindings for *query* regarding $supplement_{neg}$ instead of $supplement_Q$.
2. Create a new deductive rule $rule_{new}$ with the composition *eventComposition* within *query* as the body.
3. Give $rule_{new}$ a unique head term label and for each relevant binding add a new subterm to the head term containing the binding.
4. Add a new complex event definition for $rule_{new}$ to the AST.
5. In the original composition in the body of *rule* containing *query* substitute the event composition *eventComposition* in *query* with the head term of $rule_{new}$.

Note that the same implications and limitations that apply to negated subqueries, disjunctive compositions and groupings when dealing with nested event compositions in positive queries (as elaborated earlier) apply to ones in negated queries as well.

4.7.2.2. Existentially Quantified Queries

Let $Q = \{query_1, \ldots, query_n\}$ be a set of event or state queries in a conjunction, let $query_i \in Q$ be a positive event query containing a nested event composition, in regard to $query_i$ let $E_Q(query_i) = Q \setminus query_i$ be its so-called environment, let *head* be the head term of the deductive rule the body of which contains the conjunction, let $supplement_Q$ be the supplement to the conjunction encompassing Q, and let $supplement_{ex}$ be an existential quantification's supplement. Listing 4.7.27 shows the general conjunctive event composition pattern with existentially quantified queries using the definitions introduced above.

Listing 4.7.27: Existentially quantified $query_i$ pattern

```
DETECT
  head
ON
  and {
    query_1 ,
    ⋮
    exists { query_i } supplement_ex ,
    ⋮
    query_n
  } supplement_Q
END
```

Existentially quantified queries are considered negative since they cannot produce bindings. As such, they are treated just like negated queries, i.e. as elaborated above. One simply has to replace $supplement_{neg}$ with $supplement_{ex}$ in the explanations of section 4.7.2.1.

4.7.3. Complete Transformation

The previous sections describe how single nested event compositions are resolved, but the explanation of the entire transformation is incomplete without a description on how it is applied to an entire program's AST. The following steps describe the complete transformation for resolving nested event compositions:

1. Collect all deductive rules.
2. Consider every query containing a nested event composition, found at the top level of the body of these deductive rules and resolve it as described in section 4.7.1.2 and 4.7.2.1. Keep track of the newly created deductive rules and for their contained queries with nested event composition do this same step (two) again.

Note that as can be seen in step two the complete transformation is performed top-down. It stops when no more nested event compositions are found.

4.8. Translating Existential Quantification

Existential quantification (using the `exists` construct) is a valuable tool of Dura to limit the amount of events (or possibly states) matching a query. Such queries are called existential queries and as are lacking in Dura_C. The transformation described in this section describes how to achieve such queries in Dura_C with the existing facility of grouping and a generic translation to appropriate grouping expressions.

4.8.1. Existential Queries

Let $Q = \{query_1, \ldots, query_n\}$ be a set of atomic (i.e. flat), event or state queries in a conjunction. At the time of applying the transformation described in this section these queries are all flat because an earlier transformation has removed all nested event compositions. Note that at least part of these queries can also be negative queries, where "negative" refers to both negated queries (`not`) as well as existential queries. Although the latter type is not a negation, just like the former type of queries, it cannot produce bindings matchable by other queries in the same composition. Let $query_i \in Q$ be a positive event or state query, in regard to $query_i$ let $E_Q(query_i) = Q \setminus query_i$ be its so-called environment, let *head* be the head term of the deductive rule the body of which contains the conjunction, let $supplement_Q$ be the supplement to the conjunction encompassing Q, and let $supplement_{ex}$ be the existential quantification's supplement.

Listing 4.8.1: Existentially quantified $query_i$ pattern

```
DETECT
  head
ON
  and {
    query1 ,
    ⋮
    exists { queryi } supplementex ,
    ⋮
    queryn
  } supplementQ
END
```

Listing 4.8.1 shows the general conjunctive event composition pattern with existentially quantified subqueries using the definitions introduced above. The existential quantification's supplement $supplement_{ex}$ serves the same purpose as a negation's supplement. It relates elements within $query_i$ with positive queries in $E_Q(query_i)$ and ought to provide a time frame limitation in relation to such positive queries.

4.8.2. Translation Idea

Consider the example presented in listing 4.8.2. The entire composite query in the rule's body shall match for every event `x` during which *at least one* event `y` occured.

Listing 4.8.2: Existential query example

```
DETECT
  t{ p{ var A} }
ON
  and {
    event i: x{ p{ var A } },
    exists { event j: y{} } where { event j during event i }
  } where { var A > 10 }
END
```

Consider tables 4.8.1 and 4.8.2 and the accompanying illustration. They are a extremely crude representation of a portion of example states of the runtime system merely used for their illustrative properties. There are two overlapping instances of event x, and three instances of event y, two of which occur during the first instance of event x. Thus, due to the rule of listing 4.8.2, during the observed time only one instance of event t would be generated (cf. table 4.8.3).

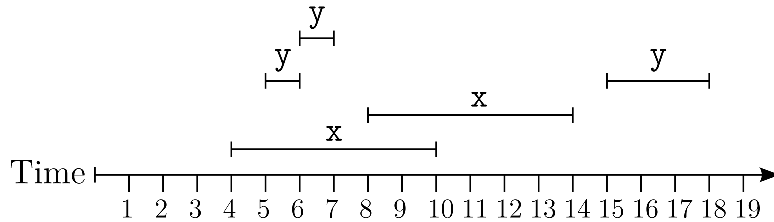


Table 4.8.1.: Event x instances

x.id	x.beg	x.end	x.p
3	4	10	30
11	8	14	28

Table 4.8.2.: Event y instances

y.id	y.beg	y.end
1	5	6
2	6	7
6	15	18

Table 4.8.3.: Event t instances

t.id	t.beg	t.end	t.p
4	4	10	30

If we moved the supplement of the existential quantification in listing 4.8.2 to the whole composition's supplement and queried for event y just like we queried for event x, there would occur two instances of event t with equal data except for ID.

However, if we additionally grouped for every identifier and every variable, i.e. all bindings, that occur in positive events this would not happen. In this case we need to group by `event i` and `var A`. Event `event j` was part of the (negative) existential quantification and is thus *not* grouped by. Listing 4.8.3 shows how this grouping is expressed.

With this code consider again the example states and instances of tables 4.8.1 and 4.8.2 and consider these tables naturally joined as presented in table 4.8.4. The selection condition `event j during event i` in the former existential quantification's supplement causes most entries to be removed as shown in table 4.8.5. In this table we can see the two matches

Listing 4.8.3: Existential query example with grouping

```
DETECT
  t{ p{ var A} }
ON
  and {
    event i: x{ p{ var A } },
    event j: y{}
  } where { event j during event i }
    group by { event i, var A }
    where { var A > 10 }
END
```

that, untreated, would cause occurrence of two instances of event `t`. However, grouping by `event i` and `var A` produces table 4.8.6. Note that the the begin and end reception times of `event i` are implicitly grouped by as well as they are, like the ID, uniquely linked to an event instance.

Table 4.8.4.: Joined tables

x.id	x.beg	x.end	x.p	y.id	y.beg	y.end
3	4	10	30	1	5	6
3	4	10	30	2	6	7
3	4	10	30	6	15	18
11	8	14	28	1	5	6
11	8	14	28	2	6	7
11	8	14	28	6	15	18

Table 4.8.5.: Selection `event j during event i` applied

x.id	x.beg	x.end	x.p	y.id	y.beg	y.end
3	4	10	30	1	5	6
3	4	10	30	2	6	7

Table 4.8.6.: Grouped by `event i` and `var A`

x.id	x.beg	x.end	x.p	y.id	y.beg	y.end
3	4	10	30	$\{1,2\}$	$\{5,6\}$	$\{6,7\}$

This illustrates the principal idea to be used for translating existential queries to groupings. However, note that the evaluation process described above is only a crude simplification of the process the abstract machine, i.e. the runtime system, would follow during runtime for such states and the given deductive rule.

4.8.3. Translation

Recall the definitions and pattern introduced in section 4.8.1 and the function Bindings (cf. algorithm 4.7.1) of section 4.7.1.1. Let *Environment* be a set of queries. The function

POSITIVE-BINDINGS described in algorithm 4.8.1 collects all bindings exposed by positive queries in an environment.

Algorithm 4.8.1: Filtering for relevant bindings.
POSITIVE-BINDINGS(*Environment*)
(1) $Q \leftarrow \emptyset$
(2) **foreach** $q \in Environment$
(3) **if** q is positive
(4) $Q \leftarrow Q \cup \text{BINDINGS}(q)$
(5) **return** Q

Let $binding_1, \ldots, binding_k \in \text{POSITIVE-BINDINGS}(E_Q(query_i))$. Then listing 4.8.4 shows the translation of the pattern presented in listing 4.8.1 to valid Dura_C code using `group by`.

Listing 4.8.4: Existential query translated

```
DETECT
  head
ON
  and {
    query_1 ,
    ⋮
    query_i
    ⋮
    query_n
  } supplement_ex
    group by { binding_1 , ..., binding_k }
    supplement_Q
END
```

This translation can be optimized so that only the first time an existential query is translated its supplement is actually inserted as $supplement_{ex}$ before the composition's supplement. Every other existential query of the *same environment* can be translated by merely adding the contents of its supplement to the existing $supplement_{ex}$.

Note that this translation could be easily extend so that Dura could allow a construct such as `at least` k where $k \in \mathbb{N}$. Let e be the event identifier within such an "at least" query. It would first be translated just like an existential query but directly after the new `group by` block the condition `count(event` e`) >=` k would be added to a (new) `where` block.

One might ask the question why the `group by` block is not appended at the very end. This might sometimes be a wise choice due to the fact that during evaluation at runtime `where` blocks in the composition's supplement might greatly reduce the amount of data in need of consideration for grouping. However, sometimes the opposite can be true as well. Since both solutions are equally viable (in regard to this runtime behavior), the reason we choose to insert the `group by` block *before* the composition's original supplement[4] is simple. In order to find out which bindings to group by we only need to look at the queries

[4] In fact it is inserted before the first element of the supplement but this is only an implementation detail.

contained in the composition. If we were to append the `group by` after the supplement we would have to consider potential additional bindings within the supplement (due to `let` and `aggregate` within the supplement) and ignore hidden bindings (due to `group by` within the supplement).

4.8.4. Complete Transformation

The complete desugaring transformation for existential queries can be achieved by the following steps:

1. Collect all deductive rules.
2. For every deductive rule collect all existential quantifications.
3. For every existential quantification translate it as described in section 4.8.3.

4.9. Replacing Aggregation Operations

In Dura, aggregation operations such as `avg(...)` or `count(...)` may be used in the head term of deductive rules as well as in `where` and `let` blocks. However, Dura_C only allows for aggregation within a dedicated construct. This section describes how such situations are handled by the desugarer.

4.9.1. Deductive Rule Pattern and Grouping in Head

Let $op \in \{\texttt{and}, \texttt{or}\}$ be an event composition (conjunction or disjunction) operator, let $term_{head}$ be a term, let $supplement_{body}$ be the supplement to an event composition, and let $supplement_{head}$ be a rule head's supplement. This supplement is either empty or a grouping construct (`group by`). Listing 4.9.1 shows the general deductive rule pattern using the definitions introduced above. Note that grouping either in the head or body of a rule implies that $op =$ `and`.

Listing 4.9.1: Deductive rule pattern

```
DETECT
  term_head  supplement_head
ON
  op {
     ⋮
  } supplement_body
END
```

The first order of business is to move the head's supplement to the end of the rule's body (cf. listing 4.9.2). This is motivated by the fact that the head term $term_{head}$ has access only to bindings (i.e. variables, identifiers) that are visible (or matchable) at the end of the rule's body, i.e. after the last element of $supplement_{body}$. So, grouping in the head and then using the grouping's result in the head term is the same as grouping after the last supplement element in $supplement_{body}$ and then using the grouping's result in the head term.

Listing 4.9.2: Deductive rule pattern with moved head supplement

```
DETECT
  term_head
ON
  op {
     ⋮
  } supplement_body
    supplement_head
END
```

4.9.2. Handling Aggregations in Body Supplement

A deductive rule's body's supplement $supplement_{body}$ may contain `where`, `let`, and `group by` blocks. A `group by` block may contain an additional `aggregate` block which, similar to a `let` block contains variable definitions. These variable definitions have the form `var` $name_{var}$ `=` $aggregationOp$`(var` $name_{groupedVar}$`)` where $name_{var}$ is the name of a new variable to be defined, $aggregationOp$ is an aggregation operator and $name_{groupedVar}$ is the name of a variable that was grouped, i.e. one that was *not* grouped *by* in the `group by` the `aggregate` block belongs to. A supplement can be seen as a list, i.e. order is important. After a variable has been defined and defined in an `aggregate` block it is visible to the following supplement elements.

Assume that $letOrWhere$ is either a `where` or a `let` block which contains an aggregation operations, e.g. `where{ var Y > avg(var X) }`. Let $nameToAgg$ be an initially empty mapping between names and aggregation operation. For $letOrWhere$ we can resolve all contained aggregation operations as follows:

1. Collect all aggregation operations $agg_1, \ldots, agg_n$ within $letOrWhere$.
2. For every agg_i replace it in $letOrWhere$ with a new uniquely named variable var_i with name $name_i$ and add $(name_i, agg_i)$ to $nameToAgg$.
3. Find the last `group by` block $groupBy$ before $letOrWhere$ within $supplement_{body}$.
4. If $groupBy$ has an attached `aggregate` block, let $aggregate$ be this block. Otherwise attach a new `aggregate` block $aggregate$.
5. For every $(name_i, agg_i) \in nameToAgg$ add `var` $type_i$ $name_i$ `=` agg_i to $aggregate$, where $type_i$ is the type of agg_i. Depending on the actual aggregation operation this type is known, e.g. an averaging operation will have a decimal type, or it can be inferred from the rule's body.

To make a rule valid for Dura$_C$ these steps have to be performed for every $letOrWhere_i$ (that contains aggregation operations) in $supplement_{body}$. Listings 4.9.3 shows an example containing aggregation operations and listing 4.9.4 shows its translation.

Listing 4.9.3: Aggregation in supplement example

```
group by { var A, var B }
let      { var E = avg(var D) }
where    { var A < avg(var C) }
```

Listing 4.9.4: Aggregation in supplement example translation

```
group by  { var A, var B }
aggregate { var float _5_0 = avg(var D), var float _5_1 = avg(var C) }
let       { var E = var _5_0 }
where     { var A < var _5_1 }
```

4.9.3. Handling Aggregations in Head Term

In Dura, the leaves of a deductive rule's head term may instead of variables and identifiers also contain aggregation operations or arithmetic expressions containing aggregation operations.

The steps required to handle these constructs is very similar to the ones used for handling aggregation operations in a deductive rule's body. Let $term_{head}$ be a rule's head term, let $supplement_{body}$ be the rule's body's supplement, and let $nameToAgg$ be an initially empty mapping between names and aggregation operation. For $term_{head}$ we can resolve all contained aggregation operations as follows:

1. Collect all aggregation operations agg_1, ..., agg_n within $term_{head}$.
2. For every agg_i replace it in $term_{head}$ with a new uniquely named variable var_i with name $name_i$ and add $(name_i, agg_i)$ to $nameToAgg$.
3. Find the last `group by` block *groupBy* from the end of $supplement_{body}$.
4. If *groupBy* has an attached `aggregate` block, let *aggregate* be this block. Otherwise attach a new `aggregate` block *aggregate*.
5. For every $(name_i, agg_i) \in nameToAgg$ add `var` $type_i$ $name_i$ `=` agg_i to *aggregate*, where $type_i$ is the type of agg_i. Depending on the actual aggregation operation this type is known, e.g. an averaging operation will have a decimal type, or it can be inferred from the rule's body.

4.9.4. Complete Transformation

The complete transformation for aggregation operations in other places than allowed by Dura_C, can be described with the following steps:

1. Collect all deductive rules.
2. For every deductive rule:
   a) Move grouping in head to body as described in section 4.9.1.
   b) Handle aggregation operations in body supplement as described in section 4.9.2.
   c) Handle aggregation operations in head term as described in section 4.9.3.

4.10. Resolving Expressions in Rule Heads

After applying the transformations described in the previous sections, head terms of deductive rules may still contain arithmetic expressions within term leaves. These have to be replaced by variables that are defined appropriately in the associated rule body as described in the following.

4.10.1. Deductive Rule Pattern

Let $op \in \{\texttt{and}, \texttt{or}\}$ be an event composition (conjunction or disjunction) operator, let $term_{head}$ be a term and let $supplement_{body}$ be the supplement to an event composition. Listing 4.10.1 shows the general deductive pattern using the definitions introduced above.

Listing 4.10.1: Deductive rule pattern

```
DETECT
  term_head
ON
  op {
     ⋮
  } supplement_body
END
```

The term $term_{head}$ may contain arithmetic expressions using variables visible at the end of the rule's body. For instance, if variables of decimal type `var A` and `var B` are not hidden, the arithmetic expression `2 * (var A / var B)` is allowed in term leaves of the head term of the associated rule.

4.10.2. Resolution

Let $nameToExp$ be an initially empty mapping between names and arithmetic expressions. For $term_{head}$ (cf. listing 4.10.1) we can resolve all contained arithmetic expressions as follows:

1. Collect all arithmetic expressions $exp_1, \ldots, exp_n$ that are term leaves within $term_{head}$. This means that no subexpressions of expressions are collected.
2. For every exp_i replace it in $term_{head}$ with a new uniquely named variable var_i with name $name_i$ and add $(name_i, exp_i)$ to $nameToExp$.
3. Attach a new `let` block let_{exp} to the end of $supplement_{body}$.
4. For every $(name_i, exp_i) \in nameToExp$ add `var` $type_i$ $name_i$ `=` exp_i to let_{exp}, where $type_i$ is the type of exp_i. This type can be inferred from the expression's position in the head term and the schema of the event associated with the rule.

Listings 4.10.2 shows an example containing aggregation operations and listing 4.10.2 shows its translation.

Listing 4.10.2: Expressions in head example

```
EVENT
  q{ p{ float } }
WITH
  DETECT
    q{ p{ 2 * (var A / var B) } }
  ON
    and {
      event: r{ p{ var A } },
      event: s{ p{ var B } }
    } where { var B != 0 }
  END
END
```

Listing 4.10.3: Expressions in head example translation

```
EVENT
  q{ p{ float } }
WITH
  DETECT
    q{ p{ var _6_0 } }
  ON
    and {
      event: r{ p{ var A } },
      event: s{ p{ var B } }
    } where { var B != 0 }
      let   { var float _6_0 = 2 * (var A / var B) }
  END
END
```

4.10.3. Complete Transformation

The complete transformation for resolving expressions in deductive rule heads can be achieved by performing the following steps:

1. Collect all deductive rules.
2. For every deductive rule resolve expressions in head term as described in section 4.10.2.

5. Future Work

The goal of this thesis was to create a desugarer that translates Dura code to semantically equivalent Dura_C code. As has been foreshadowed in chapter 1 and chapter 2, this goal has only been partly achieved. While we believe that the implemented translations are covering the most important features of Dura, time constraints prevented us from completing the implementation of the entire feature set of Dura. This chapter will analyze in what regard the current implementation is still incomplete and how one would need to go forward in completing it. In addition, improvements to the current implementation as well as possible limitations due to the undertaken desugaring approach are discussed.

5.1. Limitations of the Implementation

The current implementation of the Dura desugarer does not offer translations for the following major Dura constructs: `WHILE ... LET ... END` statements, `DERIVE ... FROM ... END` rules, and `IF ... THEN ... ELSE ... END` statements. In the following we will give pointers for their future implementation.

As pointed out in [13] a `WHILE ... LET ... END` statement is equivalent to the set of contained deductive rules considered standalone with a small change to their body. For every rule, to the body's query the `WHILE` statement's state query is added. Additionally, a `where` block needs to filter those matches of the body's original query to those (complex) events that occur when the state is valid. One of the reasons this was not implemented yet is, that at the time of writing it was still under evaluation where these `WHILE` statements may occur in the first place. Normally deductive rules are grouped together with their corresponding event definition. This is specifically pointed out as advantageous in [5]:

> "[Grouping] similar rules makes large programs clearer and easier to understand. One can quickly get an overview of how a certain event type is derived by looking at a small and particularly coherent part of the program. Moreover, because rules are clustered in a small region of the program instead of being scattered throughout the whole program, rules that derives a certain event are easier to find and to compare."

When regarding `WHILE` statements there is bound to occur a conflict with this advantage. Allowing `WHILE` statements at the top level of Dura programs will inevitably nullify the aforementioned advantage since deductive rules could occur independent of an event definition. The advantage of allowing `WHILE` statements at the top level would be, that one could quickly see all rules related and affected by a given state query. On the other hand, if one were to only allow `WHILE` statements as part of event definitions, one would maintain the advantage of grouped deductive rules for the same event, but would lose the advantage of

being able to quickly summarize all rules in a program affected by the given state query. This is why we would suggest to remove the `WHILE` statement altogether.

In order to translate reasoning on stateful objects in the form of `DERIVE ... FROM ... END` rules it is highly likely that reactive rules with queries for stateful object changes, combined with (internal) action executions for creating and modifying stateful objects provide all the tools required. Due to the introduction of schemata, the names of these (stateful object modification) actions and events entailed by changes to stateful objects as introduced in [13] need to be respecified. This will most likely follow a similar approach as was pursued for the names of events entailed by action execution.

Handling conditional `IF ... THEN ... ELSE ... END` statements within complex action rules and reactive rules can most likely be achieved by a combination of reactive rules and an added instance synchronization term like the `_composition` term used for the current translation of complex action rules. Semantic analysis, i.e. checking executability, will pose a considerably more difficult task than it currently is. It might even turn out that reliably determining this statically at compile time is not possible.

Besides the syntactic constructs mentioned above there are a few minor ones as well. Currently action execution constraints to be used in action compositions can only have a fixed form. A future implementation will provide automatic transformation of arbitrary constraints into the required standard form. Also, although the temporal relations `before` and `after` are not yet supported in the `where` block of action compositions, they can be easily implemented by translation to temporal constrains.

Also, due to the introduction of the extended `concurrent` action composition, the `and` and `or` compositions need to be reevaluated. As they are described in [13] these composition types can be translated to `concurrent` compositions with an appropriate specification of success in the `succeeds on` block.

With the aforementioned limitations in mind, it is apparent that ultimately the current desugarer can only translate a sublanguage of Dura to Dura_C. Regarding the implementation of the desugarer in general, although it was implemented and tested, e.g. the resulting AST was checked and validated to the best of our ability, more extensive testing and validation of output code in a runtime environment would be desirable. Due to the lack of formal specifications and semantics for Dura, formal validation has not been pursued as of yet.

5.2. Possible Extensions and Optimizations

While priority for future work lies on eliminating the current limitations of our implementation, during the work on this thesis several ideas for extensions and optimizations emerged. These will be presented in the following.

Currently the execution (or rather initiation) of actions as part of a complex action can only be constrained with a conjunction of a certain type of temporal constraints. This is sufficient in order to specify the order of execution of actions in relation to time points, as well as minimum waiting times between executions. However, it might be desirable to make the execution of an action depend on the positive outcome of only *one* of several other actions.

For instance, let a, b, c, d be actions. With composition nesting it is currently possible to state that action c shall be initiated after either a or b ended successfully (cf. listing 5.2.1), but additionally specifying that d shall be initiated after a ended successfully is not possible. Such execution constraining could be translated to reactive rules similar to the currently implemented translation. However, semantic analysis will require adaption to the added uncertainty. Furthermore, the exact syntax will require specification. Possibly, in a future version of Dura constraining will not be done in the `where` block of an action composition, but in `where` blocks local to the actions to be initiated so that one can immediately see on what a particular action initiation depends.

Listing 5.2.1: Initiation of c depending on success of a or b

```
concurrent {
  action x: concurrent {
    action a: ...,
    action b: ...
  } succeeds on {
    or {
      event: action a$succeeded{},
      event: action b$succeeded{}
    }
  },
  action c: ...
} where { end(action x) - begin(action c) <= 0 min }
END
```

In the current implementation success or failure of an external or complex action can be specified by special, explicit event queries in the `succeeds on` or `fails on` block. Oftentimes, it might only be necessary to check whether certain time constraints have been met. For these cases one could allow that the `succeeds on` or `fails on` blocks do not contain an explicit query, but instead a list or formula of conditions, e.g. `action b during action a` or `end(action b) - begin(action b) <= 15 min`. This could be translated to explicit queries by a desugaring transformation. Besides offering a concise specification mechanism, this extension would allow for further extension of the semantic analysis. While it is hard or not possible at all to statically check whether the explicit queries in the success or action specifications cause the complex action not to be executable, it might be possible to do so for the aforementioned, limited success or failure conditions. Checking consistency for the temporal constraint network [9] represented by these conditions in combination with the execution constraints seems like a viable path to pursue. In order to handle strict temporal constraints it might be necessary to adapt the consistency check methods described in Dechter et al. [9] in a similar fashion as was done in Bry et al. [7] for handling strict as well as non-strict temporal constraints.

Apart from extensions, some of the currently present desugaring transformations could be further optimized. For instance, resolving aggregation operators and arithmetic expressions in the head of deductive could be improved to result in less Dura_C code and variable usage by collecting all semantically equivalent expressions and use shared variables for them.

Lastly, we would like to point out a possible optimization for handling execution constraints and waiting in the translation of complex action rules. Let a_{init}, b_{init}, c_{init} be action initiation time points and b_{init} shall occur 2 minutes after a_{init} and c_{init} shall occur 2 minutes after b_{init}. Additionally, c_{init} shall occur 2 minutes after a_{init}. Obviously this last constraint is superfluous as the other constraints imply that this it is always satisfied. However, in the current implementation this constraint is also considered and maintained. In order to detect and remove irrelevant, superfluous constraints the notion of so-called dominant edges in a temporal constraint network and their detection as presented in Muscoletta et al. [19] seem to offer promising ideas.

5.3. Shortcomings of the Desugaring Approach

All in all we consider the desugaring approach for Dura to be a success. However, it also comes with a few shortcomings. At the root of these shortcomings lies the natural fact that for implementing the features of Dura only Dura_C constructs may be used. This proves difficult in the following three situations.

For one, the restrictions on variable usage when nesting event compositions can currently not practicably be lifted. It is very likely that this will remain as such due to underlying restrictions by TSA. This means that in order to tackle these restrictions at the desugaring level not only TSA but also Dura_C would have to be extended somehow.

Another issue is apparent when considering how we currently ensure single occurrence of the success and failure events entailed by action execution. For an elegant and less convoluted solution than the current one it would be desirable to somehow be able to annotate queries in Dura_C deductive rules to achieve the same effect.

Likewise, for the current translation of complex rules, blocking of reactive rule firings is achieved by the inbuilt facility of negation. If Dura_C (and TSA) were to offer specific constructs to express this waiting or blocking, a more elegant translation could be achieved.

6. Conclusion

In this last chapter we recapitulate the achievements of this thesis in light of the task at hand. The goal of this thesis was to design and implement a translation from Dura to Dura_{C} in order to support the overall compilation of Dura to Temporal Stream Algebra. After introducing the important features of both languages we presented the basic concepts and ideas for translation. Subsequently, we elaborated a detailed description of the Dura desugarer and its transformation. The previous chapter then revealed current and future issues and considerations related to the desugarer and the compilation project.

While ultimately we did not implement the translation of Dura's entire feature set, we accomplished to tackle the challenge of translating substantial aspects of complex actions. Furthermore, a sizable amount of (other) syntactic sugar of Dura has been analyzed for which we found appropriate translations. In chapter 5 we also pointed out possible approaches for completing the desugarer and for handling most of the unresolved issues.

Some features of Dura were not covered within the scope of this thesis. Most notably these include conditional statements as part of complex actions (`IF ... THEN ... ELSE ... END`) and derivative rules for stateful objects (`DERIVE ... FROM ... END`). Yet, hopefully, the translation concepts we described will offer valuable inspiration and guidance to whomever realizes their translation in the future.

With the aforementioned achievements and imperfections in mind, we believe that the results of this thesis advance the Dura compilation project as a whole and constitute a valuable contribution to the EMILI project.

A. Complex Action Rule Example

Consider the complex action rule in listing A.0.1. Its desugaring produces the reactive rules in listings A.0.2, A.0.3, and A.0.4.

Listing A.0.1: Complex action rule for action `a`

```
FOR
  action i: a{}
DO
  concurrent {
    action j: b{},
    action k: b{},
    action l: c{}
  } where {
    end(action j) - begin(action k) <= -3 min,
    end(action k) - begin(action l) <  -1 min
  }
END
```

How the desugared translation would behave during runtime is shown by the (simplified) example in figure A.0.1. In its upper part the translation is shown again in schematic form. Action instances and execution calls are represented by rectangular blocks with contents name, ID, parent and child index. In the lower part an exemplary execution of action `a` is shown, caused by a rule firing not specified in this simplified excerpt of the runtime system's state. Due to the synchronization methods employing the parent and child index data the action `b` instance with system ID 5 does not disturb the correct execution flow of the actions specified in the complex action rule for action `a`.

Listing A.0.2: Reactive rule for `action j: b{}`

```
ON
  and {
    event i_begin: a$initiated{
      payload{
        id{ var _1_0 }
      }
    }
  }
DO
  concurrent{
    action: b{
      _composition{
        child-index{ 0 },
        parent-action-id{ var _1_0 }
      }
    }
  }
END
```

Listing A.0.3: Reactive rule for `action k: b{}`

```
ON
  and {
    event i_begin: a$initiated{
      payload{
        id{ var _1_0 }
      }
    },
    event j_end: b$succeeded{
      payload{
        _composition{
          child-index{ 0 },
          parent-action-id{ var _1_0 }
        }
      }
    },
    not {
      event k_begin: b$initiated{
        payload{
          _composition{
            child-index{ 1 },
            parent-action-id{ var _1_0 }
          }
        }
      }
    } where { end(event k_begin) - end(event j_end) < 3 min }
  }
DO
  concurrent{
    action: b{
      _composition{
        child-index{ 1 },
        parent-action-id{ var _1_0 }
      }
    }
  }
END
```

Listing A.0.4: Reactive rule for `action l: c{}`

```
ON
  and {
    event i_begin: a$initiated{
      payload{
        id{ var _1_0 }
      }
    },
    event k_end: b$succeeded{
      payload{
        _composition{
          child-index{ 1 },
          parent-action-id{ var _1_0 }
        }
      }
    },
    not {
      event l_begin: c$initiated{
        payload{
          _composition{
            child-index{ 2 },
            parent-action-id{ var _1_0 }
          }
        }
      }
    } where { end(event l_begin) - end(event k_end) <= 1 min }
  }
DO
  concurrent{
    action: c{
      _composition{
        child-index{ 2 },
        parent-action-id{ var _1_0 }
      }
    }
  }
END
```

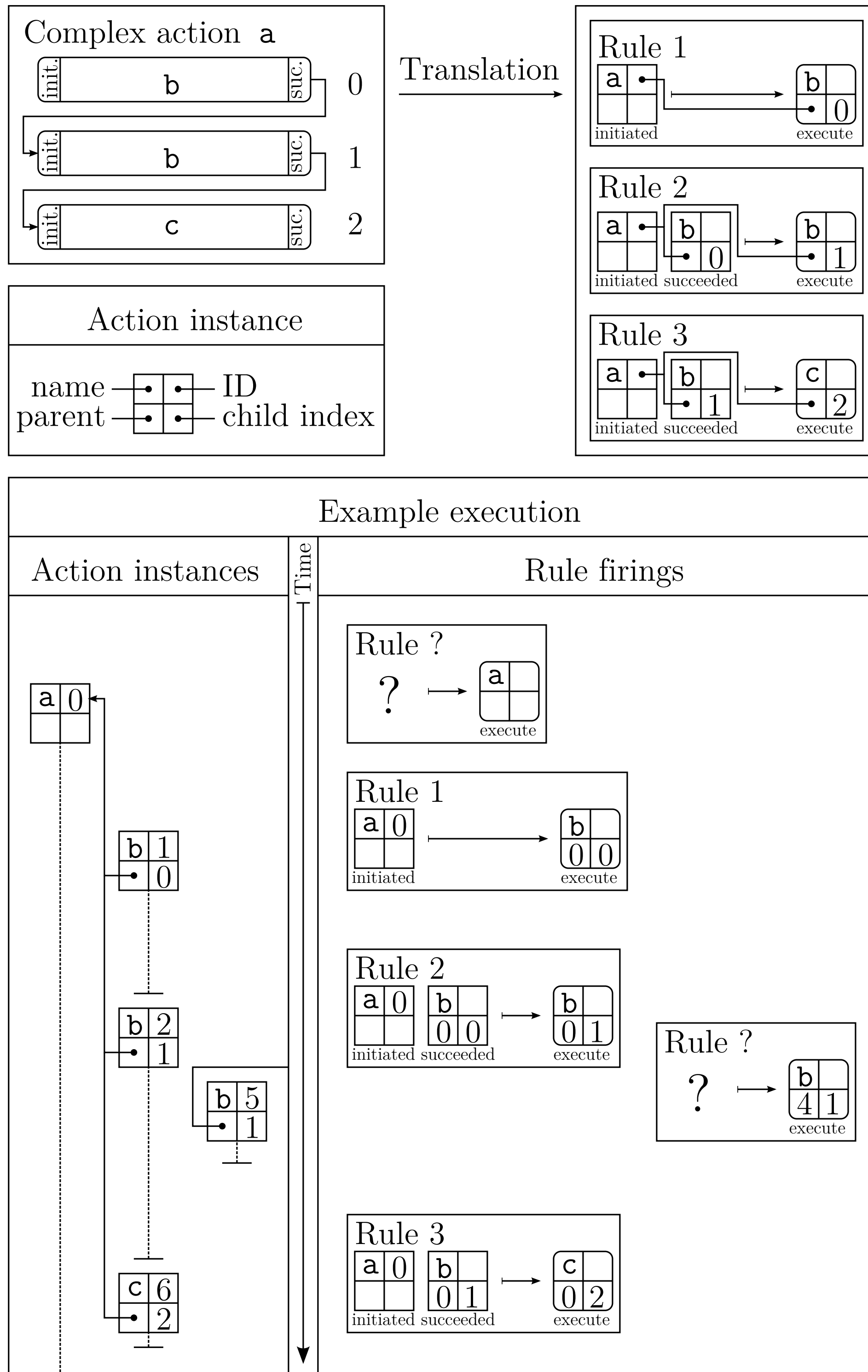


Figure A.0.1.: Action execution synchronization example

B. Nested Event Composition Example

Consider the event definition of event `d` with deductive rule in listing B.0.1. Its desugaring produces the new event definitions in listings B.0.2 and B.0.3 and changes the original deductive rule as presented in listing B.0.4.

Listing B.0.1: Event definitions and deductive rule for event `d`

```
EVENT
  a{
    p{ int },
    q{ string }
  }
END

EVENT
  b{ p{ string } }
END

EVENT
  c{
    p{ int },
    q{ float },
    r{ string }
  }
END

EVENT
  d{ p{ float } }
WITH
  DETECT
    d{ p{ var A } }
  ON
    and {
      event: a{ q{ var B } },
      event: and {
        event i: b{ p{ var B } },
        event j: c{ q{ var A } },
        event k: or {
          event: a{ p{ var C } },
          event: c{ p{ var C } }
        }
      } where { var C > var B }
    }
  END
END
```

Listing B.0.2: Event definition and deductive rule for event `_3_0`

```
EVENT
  _3_0{
    _3_0_0{ string },
    _3_0_1{ float }
  }
WITH
  DETECT
    _3_0{
      _3_0_0{ var B },
      _3_0_1{ var A }
    }
  ON
    and {
      event i: b{ p{ var B } },
      event j: c{ q{ var A } },
      event k: _3_1{ _3_1_0{ var C } }
    } where { var C > var B }
  END
END
```

Listing B.0.3: Event definition and deductive rule for event `_3_1`

```
EVENT
  _3_1{ _3_1_0{ int } }
WITH
  DETECT
    _3_1{ _3_1_0{ var C } }
  ON
    or {
      event: a{ p{ var C } },
      event: c{ p{ var C } }
    }
  END
END
```

Listing B.0.4: Event definition and changed deductive rule for event `d`

```
EVENT
  d{ p{ float } }
WITH
  DETECT
    d{ p{ var A } }
  ON
    and {
      event: a{ q{ var B } },
      event: _3_0{
        _3_0_0{ var B },
        _3_0_1{ var A }
      }
    }
  END
END
```

Bibliography

[1] EMILI Project Website. `http://emili-project.eu`. Retrieved on November 28th, 2011.

[2] H. Abelson and G. J. Sussman. Structure and interpretation of computer programs (online book). `http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-10.html#footnote_Temp_17`. Retrieved on November 26th, 2011.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison Wesley, 1986.

[4] S. Brodt, S. Hausmann, and F. Bry. Reactive rules for emergency management. Research report, PMS-FB-2010-14, Institute for Informatics, University of Munich, 2010.

[5] S. Brodt, S. Hausmann, and F. Bry. Implementation. Research report, PMS-FB-2011-14, Institute for Informatics, University of Munich, 2011.

[6] S. Brodt, S. Hausmann, F. Bry, O. Poppe, and M. Eckert. A survey on it-techniques for a dynamic emergency management in large infrastructures. Research report, PMS-FB-2010-13, Institute for Informatics, University of Munich, 2010.

[7] F. Bry and M. Eckert. On static determination of temporal relevance for incremental evaluation of complex event queries. In *Distributed event-based systems, Proceedings of 2nd International Conference on Distributed Event-Based Systems, Rome, Italy (1st–4th July 2008)*, volume 332, pages 289–300. ACM, 2008.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[9] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artif. Intell.*, 49:61–95, May 1991.

[10] M. Eckert. *Complex Event Processing with XChangeEQ: Language Design, Formal Semantics and Incremental Evaluation for Querying Events.* Dissertation/Ph.D. thesis, Institute of Computer Science, LMU, Munich, 2008. PhD Thesis, Institute for Informatics, University of Munich, 2008.

[11] GNU Prolog Manual page on DCGs. `http://www.gprolog.org/manual/html_node/gprolog041.html`. Retrieved on November 10th, 2011.

[12] G. Haddow, J. Bullock, and D. Coppola. *Introduction to Emergency Management.* Elsevier Science, 2010.

[13] S. Hausmann, S. Brodt, and F. Bry. Dura – concepts and examples. Research report, PMS-FB-2011-1, Institute for Informatics, University of Munich, 2011.

[14] S. Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge Univ Press, 2003.

[15] P. Morris and N. Muscettola. Temporal dynamic controllability revisited. In *Proceedings of the 20th national conference on Artificial intelligence - Volume 3*, pages 1193–1198. AAAI Press, 2005.

[16] P. Morris, N. Muscettola, and T. Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 1*, pages 494–499, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[17] P. H. Morris and N. Muscettola. Execution of temporal plans with uncertainty. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 491–496. AAAI Press, 2000.

[18] N. Muscettola, P. Morris, B. Pell, and B. Smith. Issues in temporal reasoning for autonomous control systems. In *Proceedings of the second international conference on Autonomous agents*, AGENTS '98, pages 362–368, New York, NY, USA, 1998. ACM.

[19] N. Muscettola, P. Morris, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *In Principles of Knowledge Representation and Reasoning*, 1998.

[20] N. Muscettola, I. Tsamardinos, and L. Hunsberger. Temporal reasoning for planning, scheduling and execution in autonomous agents (online tutorial). `http://www.cs.vassar.edu/~hunsberg/__papers__/hunsberger-tutorial-slides-aamas2005.pdf`. Retrieved on November 12th, 2011.

[21] T. Parr. *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*. Pragmatic Bookshelf, Raleigh, NC, 2007.

[22] L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[23] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the 10th international joint conference on Artificial intelligence - Volume 2*, pages 1039–1046, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[24] N. Seifert and M. Bettelini. Deliverable d3.1: Use cases requirements analysis and specification (main report). Technical report, ASIT, 2010.

[25] N. Seifert, M. Bettelini, and S. Rigert. Deliverable d3.2 – annexe a: Emergency management and rules in control systems of critical infrastructures. Technical report, ASIT, 2011.

[26] J. Stedl and B. Williams. A fast incremental dynamic controllability algorithm. In *In ICAPS Workshop on Plan Execution: A Reality Check*, 2005.

[27] L. Sterling and E. Shapiro. *The Art of Prolog (2nd ed.): Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1994.

[28] F. A. Turbak and D. K. Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008.

[29] T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: From consistency to controllabilities. *Journal of Experimental and Theoretical Artificial Intelligence*, 11:23–45, 1999.

[30] V. A. Ziparo, L. Iocchi, D. Nardi, P. F. Palamara, and H. Costelha. Petri net plans a formal model for representation and execution of multi-robot plans. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 1*, AAMAS '08, pages 79–86, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.