INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Projektarbeit

Instantiating Hierarchical Timed Automata for Modeling Complex Event Processing Systems

Sandro Gießl

Aufgabensteller:

Prof. Dr. François Bry

Betreuerin:

Olga Poppe

August 31, 2011

Acknowledgements

I would like to thank Prof. Dr. François Bry, Simon Brodt, Steffen Hausmann, and especially my supervisor Olga Poppe for discussing and supporting my work. Olga Poppe contributed much to the elaboration of the concepts of IHTA and writing of this report. I enjoyed getting to know the inner life of the research unit and being regarded as a valued part of it.

Abstract

The application logic of many information systems often follows a given workflow enforcing certain orders and dependencies of events. This knowledge can be used to make event processing more efficient. To this end Instantiating Hierarchical Timed Automata (IHTA) capturing the knowledge will be formally defined in this work. Later, constraints will be derived from IHTA and used by the semantic query optimisation algorithm.

IHTA are nondeterministic automata the states of which correspond to the application states and the edges between them are labeled with event queries and/or temporal constraints. The labels of IHTA are more expressive than that of Timed Automata initially introduced in [4], for the following reasons. First, the event queries of IHTA allow access to event data. Second, the temporal constraints of IHTA are expressed on the beginning and the end of the occurrence time of matched events (and not on local clocks).

Each state of IHTA is either atomic or non-atomic, i.e. is IHTA itself, so that an arbitrary level of abstraction can be reached. The idea of a hierarchy of models is not new. Hierarchical Timed Automata [27] and Statecharts [44] are examples of such models. However, only a fixed number of concurrent processes can be modelled by them. IHTA extend them by *instantiation* allowing an arbitrary number of concurrent processes (specified by the same non-atomic state) to run at the same time.

Contents

Chapter 1

Motivation

1.1 Workflow in Event-based Applications

Events usually arrive not accidentally on event streams but follow the predefined application workflow and/or obey to the physical laws. As an example for an event-based application with a predefined workflow, consider an online auction use case.

Example: An arbitrary number of auctions may take place concurrently. Each of them runs according to the rules described in the following. In an auction, only enrolled users are allowed to place bids for an item. Bidders are enrolled during the first 20 minutes of an auction. During the bidder enrollment, users may optionally undergo a registration step and then authenticate in order to enroll themselves for the auction. Afterward, if less than two bidders are enrolled, the auction closes. Otherwise, the auction continues and at least one item is offered to the bidders, one item at a time. For each item, bidders may place their bids with a price which must be higher than the price of the previous bids. After 30 seconds without bids and hammer beats, there is a new hammer beat. When three hammer beats took place subsequently, i.e. when there was no bid between these hammer beats, further bids are forbidden. If there was at least one bid, the item is sold to the bidder who placed the bid with the highest price. When an item has been presented, further items may be offered until the auction terminates.

As the example shows a workflow can be rather complicated. In order to use this application-specific knowledge for optimization of event queries (which is the actual purpose) the knowledge has to be expressed in an appropriate way. The following section discusses the requirements to a formalism for application semantics and motivates them by the above example.

1.2 Requirements to a Formalism for Application Semantics of Event-based Systems

A formalism for application semantics of event-based systems should:

1. be stateful,
2. express cardinality and functional dependencies between events and states,
3. be modular,
4. represent an arbitrary number of concurrent processes,
5. be non-deterministic,
6. work with events which have the same occurrence time,
7. be approximate, and finally,
8. be visual and well readable.

These requirements are motivated in this section. The actual challenge of this work is to combine the above features in one model.

1.2.1 States

Application semantics is not only dependent on the recent event but rather on the history of many previous events: For example, in an auction it is not sufficient to consider only one hammer beat to decide if the item is sold or if further bids are allowed. Instead, for an accurate decision at least the last three events of an auction have to be considered, "counting" subsequent hammer beats. However, hammer beats only serve for a simple example here. Often, there are *multiple* sequences of events which lead to a certain condition and these sequences are not only *complex* but their *duration* or *length* might be *unbounded*. Therefore, specifying these conditions by manually describing sequences of events is tedious or even impossible. To overcome this, states have to be supported by a formalism for application-specific knowledge. A state can be understood as an abstraction of events received on an event stream so far.[1] There might be various application states in the online auction, e.g. "an item is currently offered" or "no further bids for an item are possible".

[1] However in this work, a state is not considered as a history of all past events like for example in Transaction Logic [14].

When events arrive, their treatment depends on the application states during which they occur. Besides, states allow for automatic termination of queries and garbage collection of irrelevant events, i.e. events which cannot contribute to an answer for a query (any more). Constraints on event streams are usually valid during certain states and not always. Hence, event queries and constraints should be formulated in the context of application states. This is more concise and less repetitive than expressing time windows or other temporal conditions for the evaluation time of queries and the validity time of constraints. Note that the states during which events queried in query or constraint bodies arrive are known and can be determined automatically. In such cases, event queries and constraints do not even have to be expressed in the context of states explicitly.

Since an arbitrary number of processes (e.g. auctions) can take place at the same time and each of the processes has its own current state, there is a need to relate each running process to its current state. To this end, states could carry data (e.g. an auction identifier). Such a formalism would be more expressive than classical automata such as nondeterministic finite automata [65].

1.2.2 Cardinality and Functional Dependencies of Events and States

As motivated in Section 1.1, in many CEP applications, events follow specific workflows which can be rather complex. This in particular implies involved causal and temporal relationships. The following examples illustrate the relations between events but similar relations exist between states as well as between events and states.
Examples for causal relations: If at least two bidders are enrolled for an auction at least one item is presented in the auction. If there is at least one bid for an item, the item will be sold, i.e. there are at least three hammer heats and exactly one sell of the item.
Examples for temporal relations: In an auction, bids for an item happen only after the item has been presented. After a bid, there must be a subsequent bid within less than 30 seconds or a hammer beat after exactly 30 seconds.

Remember that an arbitrary number of auctions may take place at the same time and an arbitrary number of items is presented during an auction. In order to relate each event to a particular auction and a particular item, events carry data such as an auction identifier or an item identifier. This implies numerous functional dependencies between the data of different events.
Examples for functional dependencies between event data: If an item with a particular item identifier is presented only once in all auctions, an auction identifier is functionally

dependent from an item identifier in all events and states.

Besides functional dependencies, cardinality of events (i.e. the number of events during time intervals) are implied by a complex workflow.
Examples for cardinality relations: There is exactly one item description for each item. There are no or at least three hammer beats for each item.

Note that a formalism for application semantics must be able to express an arbitrary combination of the above relations between events and states because they usually hold concertedly as the above examples show.

1.2.3 Modularization

Many workflows can be divided into independent sequential or concurrent processes. For example, in an auction, there is an arbitrary number of concurrent bidder enrollment processes followed by an arbitrary number of sequential item offer processes. This motivates the idea of splitting models into readable and manageable modules representing subprocesses.

Indeed, application processes can often be described from different perspectives. Describing the workflow of an auction in terms of bidder enrollments and item offers is a high-level description compared to the specific details of bidder enrollments and item offers which are low-level descriptions. A specific process with a low-level description (e.g. a bidder enrollment) can be regarded as subprocess of another high-level process (e.g. an auction).

A process and an arbitrary number of its subprocesses can (but do not have to) run simultaneously. If a process runs it is possible that none of its subprocesses are running. But if at least one subprocess is active its respective (super) process must also be running, e.g. an item can only be presented during an auction.

A high-level process can be understood as an abstraction of its subprocesses. Supporting this abstraction in the formalism is important to reduce complexity, to increase readability, and to support stepwise refinement (i.e. beginning with a high-level specification and iteratively extending it with low-level details). Other benefits of modularization are reusability and (ex-)changeability of modules without affecting others.

1.2.4 Arbitrary Number of Concurrent Processes

As mentioned above an unbounded number of concurrent processes (e.g. auctions or bidder enrollments within an auction) must be expressed by the formalism. Each concurrent process has its own current state and runs relatively independently from other concurrent processes on the same description level (aside from certain synchronization points such as the end of all bidder enrollment processes after 20 minutes since the auction beginning have passed).

Each concurrent process can arbitrarily change its state within its state space. Without support for concurrency, this behavior could be simulated by a higher-level process description to a limited extent as illustrated by the following example. Consider two concurrent processes with the state spaces $\{a, b, c\}$ and $\{d, e, f\}$ respectively. Their current states can be simulated by nine states in the description of a higher-level process: (a, d), (a, e), (a, f), (b, d), (b, e), (b, f), (c, b), (c, e), and (c, f). This would quickly result in an unreadable model of a *fixed* number of concurrent processes. Modeling an arbitrary number of concurrent processes is not possible with such an approach. Approaches like Hierarchical Timed Automata [27] and Statecharts [44] which are similar to the approach discussed in this paragraph will be discussed in Chapter 2.

One possible solution of this problem is a formalism in which the specifications of processes are modeled by low-level descriptions which, in higher-level descriptions, are marked to be concurrent. Concurrent processes must start, change their states, and terminate dynamically during the run of the automaton.

1.2.5 Non-determinism

In contrast to the auction use case, many event-driven applications are non-deterministic, i.e. more than one state is reachable from a state. This feature must be supported by the model of application semantics. There are two kinds of non-determinism [45]:

1. Don't care non-determinism (also called conjunctive non-determinism) means that all choices will lead to a successful search, so we "don't care" which one we take.

2. Don't know non-determinism (also called disjunctive non-determinism) means that some of the choices will lead to a successful search, but we "don't know" which one a priori. There are two possibilities to treat the problem:

   (a) Either we guess one of the choices, check whether it is right, and stop guessing as soon as we have found a successful choice (like Non-deterministic Finite

State Automata, see Section 2.1.1). Otherwise continue checking the remaining choices.

(b) or we follow all possible choices and drop the choices which turn out to be wrong (like Powerset construction method translating Non-deterministic Finite State Automata into Deterministic Finite State Automata, see Section 6.2.3).

Conjunctive non-determinism is not problematic because it is known beforehand that all choices are right. This is not the case by disjunctive non-determinism which is therefore more complicated. The second solution dealing with disjunctive non-determinism allows event stream verification on-the-fly, i.e. there is no need to save past events in order to test other choices. Therefore the second way of treating disjunctive nondeterminism is more preferable than the first one.

1.2.6 Other Requirements

Other requirements to a formalism for application semantics in CEP systems are the following:

- **Treatment of events with the same occurrence time**
  If the event rate is high or the granularity of the discrete time is coarse[2] it is possible that same events have exactly the same occurrence time. The formalism of application-specific knowledge must be able to treat such events.

- **Approximate specification**
  It is quite seldom that the application semantics is fully defined and completely known beforehand. Therefore a formalism capturing it should be approximate. It should be possible to describe different parts of the model at an arbitrary level of abstraction.

- **Readable visual representation**
  Readable visual representation of a complex formalism facilitates the understandability of the model, in particular by non-technical persons. There is a trade-off between readability and expressivity of a model which is one of the main challenges.

[2]The problems concerning different time models are discussed in Section 3.1.

1.3 Organization and Contributions of this Thesis

We start in Chapter 2 with the analysis of the related work with respect to the requirements to a formalism for application semantics described in Section 1.2. Chapter 3 is devoted to our understanding of the basic notions of Complex Event Processing. Chapter 4 introduces the Instantiating Hierarchical Timed Automata (IHTA) by the auction use case described in Section 1.1. Chapter 5 and Chapter 6 are devoted to the syntax and the formal semantics of IHTA respectively which are the main contributions of this work. Chapter 7 concludes this thesis and gives an outlook.

Chapter 2

Related Work

Automata, flowcharts, Petri nets, and UML behavior diagrams are visual formalisms modeling complex dynamic workflows. Even though these approaches share similar goals, they concentrate on different aspects. This chapter describes the formalisms briefly, analyses them with respect to the requirements described in Section 1.2, and compares them with the Instantiating Hierarchical Timed Automata (IHTA) we propose.

2.1 Automata

Automata (also called (Finite) State Machines) are one of the most fundamental, widely used and well-studied modeling mechanisms in computer science since the 50's. There are different kinds of automata. We start with classical Finite State Automata in Section 2.1.1 and continue with their extensions which are relevant for IHTA, in particular the addition of variables, to form Extended Finite State Machines (Section 2.1.2), the extension to work on infinite input, to form ω-Automata (Section 2.1.3), the addition of clocks, to form Timed Automata (Section 2.1.4), the addition of hierarchical (nesting) capability with or without concurrency (communication), to form (Communicating) Hierarchical Finite State Machines (Section 2.1.5), and, finally, the differentiation between universal and existentional paths, to form Alternating Machines (Section 2.1.6).

2.1.1 Finite State Automata

Finite state automata [65] consist of (a finite number of) states and transitions between them. Transitions are labeled by symbols of an alphabet. If an automaton is in a state s and the label of an outgoing transition t of s matches the next symbol of the input word,

t fires and the automaton is in the state t leads into. If each state of an automaton has at most one outgoing transition labeled with a symbol, the automaton is called *deterministic*. If there is a state with at least two outgoing transitions with the same label, the automaton is *non-deterministic*. There are some other features distinguishing these two kinds of finite state automata which are omitted hier for the sake of brevity. We refer the reader to [65]. An automaton describes the language consisting of words which are accepted by the automaton, i.e. which lead the automaton into one of its end states.

If we consider the set of all events as an alphabet and event streams as words, automata can be seen as a description (specification) of these event streams. This issue will be discussed in Chapter 3 in more detail.

There are different kinds of automata working on event streams and extending classical Finite State Automata with additional features such as temporal constraints, hierarchy of automata, concurrency, and variables. The following sections are devoted to them. Instantiating Hierarchical Timed Automata (IHTA) we propose are also an extension of classical automata in these respects. IHTA are seen as a metadata for semantic optimization of event queries. Finite state automata were used to a limited extent for this purpose in [32], [54], [69], [39]. In these approaches, however, automata represent event queries rather than the application workflow as in our approach.

2.1.2 Extended Finite State Machines

Extended Finite State Machines [24] are Finite State Machines equipped with variables which are either input, output or local. The values of variables can be updated while a transition fires. Constraints on the values of these variables can be added to transitions. Since the domain of each variable is not restricted, Extended Finite State Machines cannot be translated into ordinary Finite State Machines.

In contrast to classical Finite State Machines, Extended Finite State Machines do not represent the entire state space explicitly and are, therefore, more compact, readable, and cheaper to build. The main idea introduced in [25] is to manipulate sets of states and transitions simultaneously and represent these sets by Boolean functions, i.e. the state space is represented symbolically rather than explicitly.

IHTA use two kinds of variables, namely data variables and identifiers. Both are local input variables which are bound while matching events or states, and can be rebound (overshadowed) later. No other variables are supported. In contrast to IHTA, Extended Finite State Machines are neither timed, nor hierarchical. They do not support concurrency and were not used as metadata for semantic query optimization.

2.1.3 ω-Automata

ω-Automata [70], [58] (also called stream automata) extend Finite State Automata to work with infinite input. There are different kinds of ω-Automata: Büchi automata [20], [61], Muller automata [10], [13], Rabin automata [47], Streett automata [47], [61], and parity automata [50], [61]. There are deterministic and non-deterministic variants of each of them. These automata accept regular ω-languages (generalizing regular languages to infinite words) but differ in acceptance criteria and succinctness of representation of a regular ω-language.

Like ω-Automata, IHTA work on infinite input. However, in contrast to IHTA, classical stream automata neglect temporal aspects of event processing. Therefore, no acceptance criterion of ω-Automata agrees with the semantics of IHTA. Whether IHTA accepts input stream depends on the current time and events arrived so far (see Definition 21). ω-Automata also neglect event data. They are not hierarchical and do not support concurrency.

2.1.4 Timed Automata

Timed Automata originally introduced by Rajeev Alur and David L. Dill in [4] in 1990, are ω-Automata equipped with a finite set of clocks. A clock is a piece-wise continuous real-valued function of time that records the time elapsed since the recent reset of the clock.[1] A clock can be reset to a new value if a transition fires. Respective command is a part of the transition label. All clocks are synchronized.[2]

The edges of Timed Automata are labeled by events, temporal constraints on clocks, and reset commands on clocks. An event is an atomic symbol (i.e. events do not carry data) and a real-valued time point of occurrence associated with it. Many events may have the same time point of occurrence. A temporal constraint on a clock is a comparison of the clock value with a number. A reset command on a clock is an assignment of a new value to the clock. Timed Automata describe a timed language consisting of timed words (i.e. event streams) which are accepted by the automaton, i.e. which lead the automaton into one of its end states.

Timed Automata are used to model and reason about real-time systems such as network protocols, business processes, reactive systems, etc. They are well-studied from the perspective of formal language theory [4], [5], [3], [7], [11], [60], [36], and model checking [79], [17], [67], [46], [51], [18], [52]. Considerable amount of work has been done on the

[1] There are however Timed Automata which are based on discrete time model, e.g. [43].
[2] There are distributed Timed Automata working with asynchronous clocks, e.g. [31].

automatic inference of Timed Automata from data [73], [74], [75], [41], [40].

Timed Automata were initially developed for event stream verification, in particular, for the expression of constant bounds on the delays between events [4]. Later, Timed Automata were also used for solving scheduling problems, consider [1]. To the best of our knowledge, they were not considered as metadata for semantic optimization of event queries.

Like Timed Automata, IHTA are able to process events with the same occurrence time. In contrast to Timed Automata, IHTA work with events which carry data and have time intervals as their occurrence time. Transitions of IHTA are labeled with event queries which provide access to event data and/or temporal constraints which are defined on the beginning and the end of the occurrence time of matched events or on current time. Temporal conditions of Timed Automata which are expressed on clocks can simulate the temporal constraints of IHTA which involve the end but not the beginning (!) of the event occurrence time. Hence, transition labels of IHTA are more expressive than that of Timed Automata. Timed Automata do not support hierarchy of automata. Hierarchical Timed Automata extend Timed Automata by this feature.

2.1.5 Hierarchical Finite State Machines

It is difficult, if not impossible, to model systems of a certain size and complexity using flat automata. Most systems can be divided into relatively independent manageable and comprehensible processes. To this end, state-diagrams were extended by the notions of hierarchy, concurrency, and communication, to form the so-called Statecharts in [44] in 1987. Hierarchy or nesting means that each state of a model may be refined by another (possibly hierarchical) model. This enables viewing the description at different levels of detail, i.e. abstraction. Concurrency and communication mean that there can be multiple models running in parallel within the same state and communicating with each other by messages (events) to synchronize their behavior.

Since 1987 Hierarchical Finite State Machines have been further developed. [78] summarizes the work on such machines with and without concurrency done until 2000. Most of the results on expressiveness, complexity, and model checking come from [6], [8], [9]. Like ordinary Finite State Machines, Hierarchical Finite State Machines capture regular languages but they gain in exponential succinctness as compared to their respective flattened machines. In other words, hierarchical machines can be translated to classical Finite State Machines at an exponential cost. A concurrent (or communicating) Hierarchical State Machine combines concurrency and hierarchy in an arbitrary manner. It still

defines only regular languages. Its flattening causes in general a double exponential blow up.

A hierarchical machine satisfies a property if its respective flattened machine does. This is the usual approach of the investigation of the properties of (communicating) hierarchical machines. But of course this flattening can be avoided as done in [8].

Among hierarchical machines, Hierarchical Timed Automata (HTA) [28], [27], [26] are the approach closest to IHTA we propose. The similarities between them are the following. First, both models support hierarchy of automata and are therefore modular (which implies readability, extensibility, (ex-)change, and reuse of modules as well as means for abstraction). States of (I)HTA which are (possibly hierarchical) automata themselves are called non-atomic. For two arbitrary states s_1 and s_2 of (I)HTA one of the following holds: s_1 and s_2 are disjoined, s_1 is completely within s_2 or s_2 is completely within s_1.[3] Second, both models capture numerous complex temporal and causal relations between events and states.

However, IHTA can be seen as both an extension and a restriction of HTA in the following respects. Like all hierarchical machines we are aware of, HTA allow for modeling a fixed number of processes running concurrently. However in real life applications their number is often unbounded. Therefore, IHTA extend HTA by the possibility of modeling an arbitrary number of concurrent processes: On the beginning of a new process, the workflow of which is represented by a non-atomic state, the state is instantiated. An unbounded number of processes which run concurrently is expressed by multiple instances of the same non-atomic state existing at the same time. This feature of IHTA is, to our knowledge, new and not present in other formalisms. IHTA are therefore more expressive than hierarchical machines. The price we pay for the expressiveness is that IHTA cannot be flattened. In other words, the results of the investigation of the properties of hierarchical machines are not applicable to IHTA in general.

Further essential additional features of IHTA compared to HTA are, first, the event queries of IHTA allow access to event data in contrast to that of HTA and, second, temporal constraints of IHTA are defined on the beginning and the end of the occurrence time of matched events or on the current time and not on local clocks like the temporal constraints of HTA which is a less expressive approach as explained in Section 2.1.4.

Other features of HTA are not adopted by IHTA because they are not needed or not applicable to achieve our goals. They are:

[3]Statecharts [44] support a different kind of state hierarchy: Two states s_1 and s_2 may overlap so that neither s_1 is completely within s_2 no vice versa. Whether such a notion of hierarchy would by a preferable extension of IHTA remains to be investigated.

1. *pseudo* states and *pseudo* transitions facilitating the hierarchy management but making the automaton bigger,

2. differentiation between *xor* and *and* superstates: If a *xor* superstate s is active then exactly one substate of s is active, if an *and* superstate s is active then all substates of s are active (IHTA support only the former kind of superstates),

3. *parallel* states expressing concurrent processes (concurrent processes are expressed by the instantiation of a non-atomic state of IHTA, whether it is necessary or preferable to instantiate different non-atomic states at the same time is an open issue),

4. *history* states permitting to resume the run of non-atomic states with the state they had before their suspension (the states of IHTA cannot be suspended),

5. invariants associated with states (this could be an extension of IHTA which is thinkable and preferable for many applications),

6. local integer variables which can be assigned during transitions and transitions can be labeled with conditions on the values of these variables (variables of IHTA are also local, they can be of an arbitrary type, and allow access to the event data; no other variables are used in IHTA),

7. prioritization of event transitions over delay transitions (this feature is simulated by negation of an event query of IHTA, consider Section 5.3 for the details),

8. channel synchronization forcing two transitions associated with a channel to be performed in an atomic step (other kind of synchronization is supported by IHTA, namely atomic termination of all instances of a non-atomic state).

2.1.6 Alternating Machines

Alternating Turing Machines [23], [22], [66], [57] are Non-deterministic Turing Machines supporting two computation modes: Existential and universal. A computation in existential mode succeeds if any choice leads to an accepting state. A computation in universal mode succeeds only if all choices lead to an accepting state.[4] More exactly: Let several configurations $\beta_1, ..., \beta_k$ be reachable from a configuration α. If the branching is existential, α leads to acceptance if at least one configuration β_i leads to acceptance. If the branching is universal α leads to acceptance if *all* configurations $\beta_1, ..., \beta_k$ lead to acceptance.

[4]Remember that classical Non-deterministic Finite State Automata support only the existential computation mode.

Alternating Turing Machines can be understood as parallel machines where the successor configurations $\beta_1, ..., \beta_k$ run independently till completion, and their result (acceptance or rejection) is combined by the configuration α. In this point Alternating Turing Machines are similar to IHTA allowing multiple instances of the same non-atomic state s to run independently from each other till completion and an instance of the superstate of s to terminate these instances of s. Termination constraints restrict the number of successful processes represented by the instances of s. If these termination constraints are absent all processes must be successful which is comparable with (but not the same as) universal computation mode. The difference between concurrent processes of IHTA and universal branching of Alternating automata is that multiple concurrent instances are joined into one branch, while join of universal branches is not defined. If there is a non-deterministic choice in a (concurrent) process, at least one of the possibilities must be accepted for the process to be successful which corresponds to the existentional computation mode.

The theoretical findings of Alternating Turing Machines are used for time- and space-classification of problems. Problems of interest are, e.g. decision problems in logic and problems concerning the existence of winning strategies in combinatorial games, involving alternating quantifiers. To our knowledge, neither classical Alternating Turing Machines nor their extension by temporal aspects (i.e. clocks) called Alternating Timed Automata [53] were used as metadata for semantic query optimization. Alternating automata neglect event data and are not hierarchical.

Summary

Summarizing the overview of different kinds of state machines and their comparison with IHTA, we would like to emphasize that automata support the notion of application state very naturally since states are their explicit components. Considerable amount of work has been done to adjust automata to the peculiarities of event processing in general and the requirements of many event-based applications in particular such as ability to work on infinite input (ω-Automata, Section 2.1.3), temporal aspects (Timed Automata, Section 2.1.4), hierarchy of models (Hierarchical Automata, Section 2.1.5), and concurrency (Hierarchical Automata, Section 2.1.5, and Alternating Machines, Section 2.1.6). To the best of our knowledge, no existing automaton model supports access to event data, i.e. non-ground terms were not used as part of transition labels.[5] Most automata have weak time aspect since event occurrence time is a time point not a time interval. Therefore, only limited temporal constraints have been considered until now. Some automata allow multiple events with the same occurrence time (e.g., Timed Automata, Section 2.1.4). No

[5]Tree automata [71], [30] work on terms represented as trees but a node of the tree triggers a transition, not the hole tree (i.e. term).

existing automaton model is able to represent an arbitrary number of concurrent processes. Communication and synchronization of a fixed number of concurrent processes is a well-investigated issue (consider Hierarchical Automata in Section 2.1.5). Most automata support non-deterministic behavior. To the best of our knowledge, automata have never been used as metadata for semantic query optimization. ([32], [54], [68], [39] transfer queries into finite automata and evaluate the automata while events arrive rather than using automata as specifications of streams).

2.2 Flowcharts

Flowcharts [12], [48] express algorithms and the flow of application processes. Mainly because of their graphical notation they are easily understandable by non-technical persons and are therefore used e.g. for workflow specification and documentation purposes. They were first presented by Frank Gilbreth in 1921. There are different kinds of flowcharts, consider [38] for one of the newer classifications. The most famous examples of flowcharts are UML activity diagrams [64] and Business Process Model [42].

Boxes, diamonds, and directed edges between them are the main components of a flowchart. Boxes represent actions. Such an action can be a call of another program or start of a process, i.e. a hierarchy of flowcharts can be simulated. Diamonds are conditions. A diamond has usually two outgoing edges depending on whether its condition is satisfied or not. Directed edges represent the process flow.

Flowcharts do not explicitly support the concept of state. States can be simulated by boxes in some cases. We are not aware of timed flowcharts, i.e. flowcharts extended by temporal conditions. The process flow represented by flowcharts is not triggered by events.[6] A flowchart is usually deterministic. A flowchart can represent a fixed number of concurrent processes. Additional components are used for this purpose, they are responsible for fork and join of concurrent processes. Flowcharts were not used for semantic query optimization.

2.3 Petri Nets

Petri nets are suitable for modeling concurrent distributed systems, in particular expressing complex synchronization schemes. They were developed by Carl Adam Petri in 1939.

[6] There is a kind of flowchart, called Event-driven Process Chain [3], [49] using the notion event in the other sense as in this work. Event is a passive element describing the circumstances under which a process works or a state a process results in.

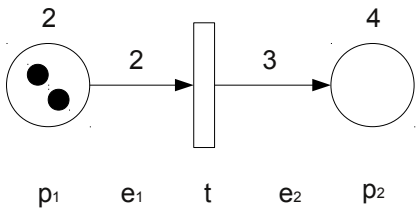

Figure 2.1: An (incomplete) Petri net

Since that time much work has been done on mathematical definition, analysis of the expressive power, decidability, complexity, reachability, liveness, and boundedness of Petri nets. [59], [29] survey this research.

A Petri net consists of transitions, places, directed edges, and tokens. Places usually represent resources. Tokens usually model concurrent processes. They move through a Petri net from one place to another illustrating resource allocation by processes. A place may have a capacity restricting the number of tokens the place may contain at most. If the capacity of a place is not restricted the place may contain an arbitrary number of tokens. Directed edges connect transitions with places and vice versa. Edges may not connect two places or two transitions directly. An edge may have a weight. Consider Figure 2.1. For the edge e_1 going out of the place p_1 the weight of e_1 specifies the number of tokens which will be removed from p_1 if the transition t fires. For the edge e_2 going into the place p_2 the weight of e_2 specifies the number of tokens which will be added to p_2 if the transition t fires. The transition t is enabled (i.e. can fire) if p_1 contains enough tokens and p_2 can include all new tokens. The transition t in Figure 2.1 is enabled, if it fires p_1 contains no tokens and p_3 contains 3 tokens.

Petri nets specify resource sharing and synchronization of an arbitrary number of concurrent processes rather than workflows of these processes (there are exceptions such as [72]). Petri nets do not support the notion of state explicitly (however, places can play this role in some cases). Transitions are not triggered by events. Therefore, Petri nets can hardly express relations between events and states.

There are extensions of classical Petri nets with respect to temporal aspects (Timed Petri nets [77], [76], [2]), hierarchy (Hierarchical Petri nets [80], [35], [56]), and both (Hierarchical Timed-extended Petri nets [37], [63]). There are different ways to introduce time into a Petri net: It can be associated with tokens, places, and/or transitions. A hierarchy construct used in Petri nets is called subnet. It is an aggegate of multiple places and transitions.

Petri nets are nondeterministic since multiple transitions can fire at the same time. To the best of our knowledge, they were not used as metadata for semantic query optimiza-

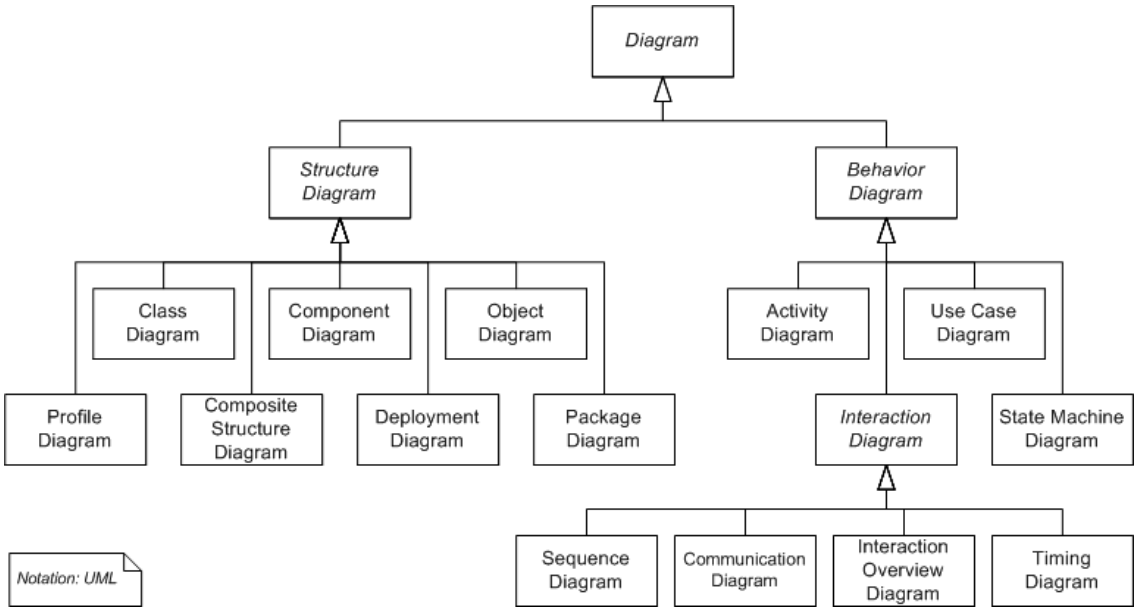

Figure 2.2: Typs of diagrams supported by UML. (Source: `http://en.wikipedia.org/wiki/Unified_Modeling_Language`.)

tion.

2.4 Unified Modeling Language

Unified Modeling Language (short UML) [16], [55], [15] had been introduced by the Object Management Group and became an international industrial standardized modeling language for the documentation, specification, visualization, and modification of object-oriented software with respect to its structural and behavioral features. Figure 2.2 contains the classification of 14 kinds of diagrams supported by UML. Since we are interested in workflow specification, only behavior diagrams are described in the following.

UML state machine diagrams are (usually deterministic) Extended Finite State Machines (Section 2.1.2) enriched by actions, output, and hierarchically nested states. Transitions between states are labeled by events and optional actions which are executed when the events trigger the transitions. A set of parameters can be associated with an event. These parameters restrict the occurrence time and the number of the respective event. Event data is neglected. An action can be update of a variable, execution of an input or an output, call of a function, or creation of an event. Two hierarchical state decompositions are supported: AND and OR. AND-decomposition means that if a composite state is active, then all its substates are also active. A fixed number of concurrent processes can be modeled by this decomposition. UML state machines also provide means for communication and synchronization of the processes. (Exclusive-)OR-decomposition means

that if a composite state is active, then exactly one of its substates is also active. As this description shows, UML state machine diagrams visualize programs rather than serve as metadata specifying event streams.

UML use case diagram provides a graphical overview of the systems functionality in terms of actors, their goals (represented by use cases) and dependencies between them to show what system functions are executed for what actor and to what purpose. The same actor can play different roles. *UML interaction diagrams* visualize the communication between the actors. In other words, UML use case and interaction behavior diagrams are not suited to specify event streams. *UML activity diagrams* are a kind of flowcharts described in Section 2.2.

Chapter 3

Basic Notions

3.1 Time

Time is represented by a linearly ordered **set of time points** $(\mathbb{T}, \leq)$, $\mathbb{T} \in \mathbb{Q}^+$ where $\mathbb{Q}^+$ denotes the set of not negative rational numbers. The **set of time intervals** is $\mathbb{TI} = \{i = [b, e] \mid b \in \mathbb{T}, e \in \mathbb{T}, b \leq e\}$. For an interval i, $b(i)$ denotes its beginning and $e(i)$ its end, i.e., $i = [b(i), e(i)]$. Note that a time interval includes its bounds. Intervals with open bounds are not considered for simplicity reasons. Time intervals with infinite bounds are also not considered since their evaluation is problematic with respect to query termination and garbage collection of event data. The notion of time intervals introduced here is that of connected time intervals. Non-connected time intervals, as needed in many applications, are not explicitly introduced here because they can easily be expressed in the formalism presented (as a disjunction of multiple time intervals).

Using continuous time represented as $\mathbb{Q}^+$ instead of discrete time represented as, e.g., $\mathbb{N}^+$ has the following advantage. If discrete time is used one have to decide how far apart two sequential discrete time points are. If the granularity of discrete time points is too coarse too many time points which are different according to continuous time fall together according to discrete time. IHTA consider all possible permutations of events with the same occurrence time (see below) which is rather expensive in general. If the granularity of discrete time points is too fine their evaluation becomes a problem.

3.2 Event Stream

Let *GroundAtoms* be a set of ground atoms of a first order language. An **event** e is tuple of $a \in \mathit{GroundAtoms}$ and $i \in \mathbb{TI}$, written a^i. a carries **event data** of e. i denotes the

occurrence time of e.

Let $Events$ be the set of events, i.e. $Events = GroundAtoms \times \mathbb{TI}$.[1] $E \subseteq Events$ is called an **event stream**.[2] All events of an event stream are totally ordered according to their occurrence time. Therefore one speaks about **sequences of events**.

We distinguish between simple and complex events. An event e is a **simple event** if it is not derived from a sequence of other events of the event stream. In this case the occurrence time of e is usually a time point. An event e is a **complex event** if it is derived from a sequence of simple or complex events. The occurrence time of e is a time interval comprising the occurrence times of all events of the sequence.

[1] $Events$ can be seen as an alphabet Σ which is not empty and possibly infinite.
[2] E can be seen as a potentially infinite word Σ^ω over Σ.

Chapter 4

Motivating Example

4.1 The Auction Use Case

As an example, let us consider the auction use case described in Section 1.1. The graphical representation of the workflow is shown in Figure 4.1. This representation is motivated by the requirements to a formalism for application semantics of event-based systems introduced in Section 1.2. The model is surely very simplified in two respects. First, event-based systems usually work on a much larger set of events. Second, events usually carry much more data. But for the purpose of the illustration of IHTA this simplified example is suitable.

4.2 Main Features of IHTA

The main features of IHTA are the following (compare Section 1.2):

1. IHTA are stateful. State changes are triggered by events and/or delays.
2. IHTA are independent from the language specifying transition labels. In Section 5.3, $\text{Datalog}^{\neg,\text{time}}$ is definied as an example of such a language used in this work. The event queries in $\text{Datalog}^{\neg,\text{time}}$ allow access to event data. The temporal constraints are expressed on the occurrence time of matched events and/or on the current time. Other kinds of constraints are not considered in this work to keep IHTA readable.
3. IHTA are non-deterministic and able to work with events with the same occurrence time.

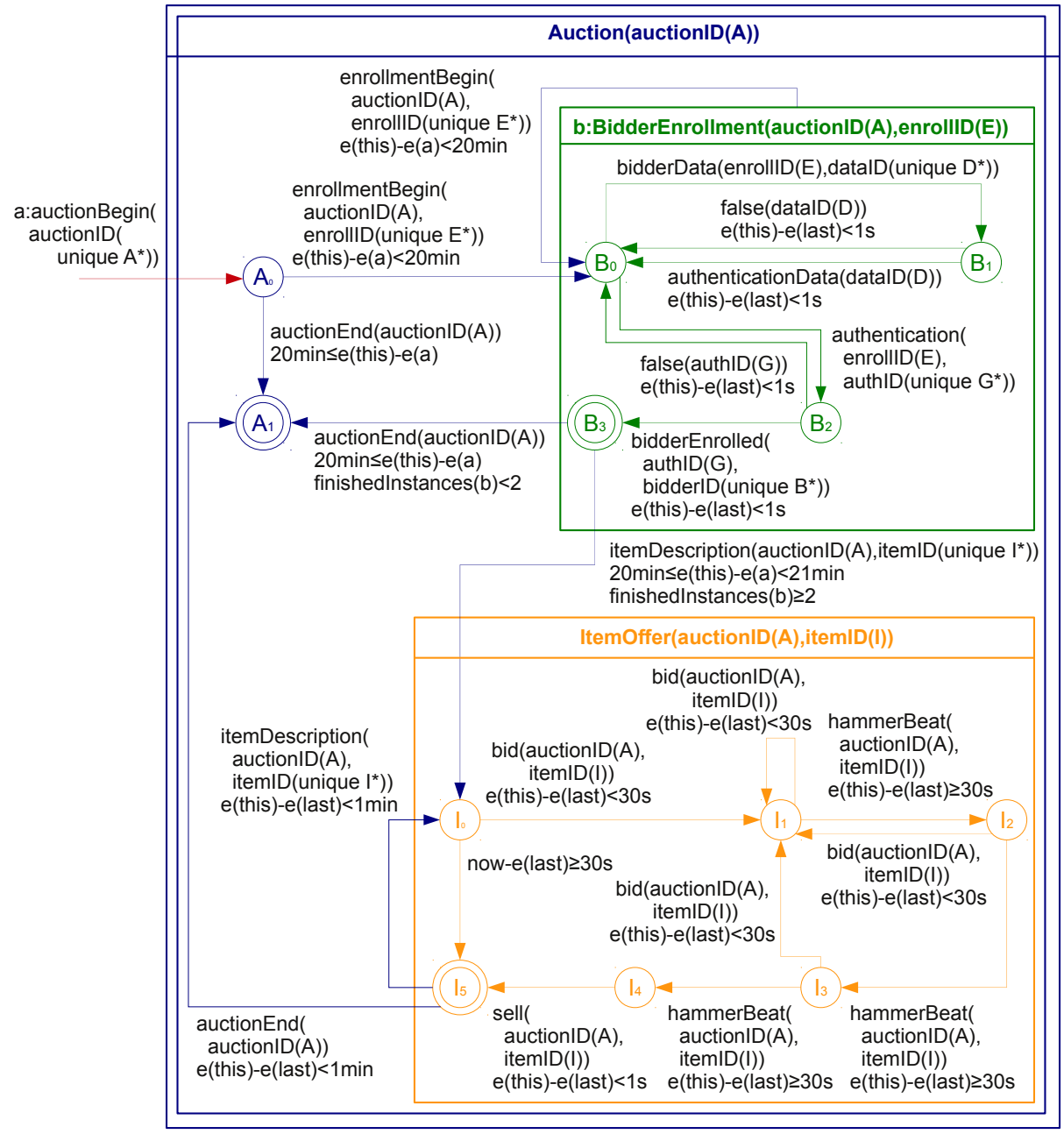

Figure 4.1: Auction modeled as IHTA.

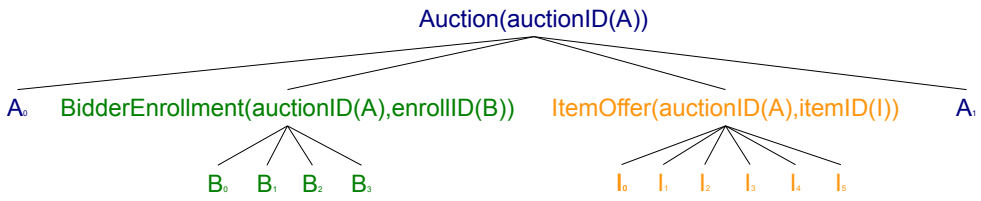

Figure 4.2: Tree of States of the IHTA in Figure 4.1

4. IHTA are modular which implies abstraction, readablility, (ex-) changeability, and reuse of the modules.

5. IHTA can represent an arbitrary number of concurrent processes by instantiation of module specifications.

Chapter 5

Syntax of IHTA

This chapter is devoted to the syntax of Instantiating Hierarchical Timed Automata (IHTA). IHTA consist of states which may contain IHTA and transitions between states. Section 5.1 describes states, Section 5.2 is devoted to transitions. Section 5.3 presents the language Datalog$^{\neg,\text{time}}$ specifying the transition labels in the examples used in this work. But IHTA are language independent in general.

5.1 States

There are four kinds of states namely atomic, non-atomic, start, and end states. They are motivated and explained in this section.

Modularization, encapsulation and the need to model an arbitrary number of concurrent processes which are motivated in Section 1.2.3 are attained by introducing hierarchy of states, i.e. the possibility to represent IHTA as a singe state of another IHTA. A state s containing IHTA $\mathcal{I}$ is an abstraction of $\mathcal{I}$ which runs while s is active. The states of the IHTA in Figure 4.1 build a tree displayed in Figure 4.2.

States containing IHTA are called **non-atomic states**. They are represented as rectangles. States which do not contain IHTA are called **atomic states** and correspond to states in classical Finite State Automata [65]. Atomic states are depicted as cycles. Each state is either atomic or non-atomic. In Figure 4.1, the non-atomic state *Auction(auctionID(A))* is a **superstate** of the non-atomic state *BidderEnrollment(auctionID(A),enrollID(E))*. The atomic state B_0 is a **substate** of the non-atomic state *BidderEnrollment(auctionID(A),enrollID(E))* which is a substate of the non-atomic state *Auction(auctionID(A))*. Trivially, a non-atomic state can be a substate and a superstate. Atomic states can only be substates.

Both atomic and non-atomic states can be either **start**, or **end**, or neither start nor end states. A start state is a state of IHTA which is entered first. For example, A_0, B_0, and I_0 are the start states of their respective IHTA in Figure 4.1. Each IHTA have at least one start state (IHTA are not deterministic). An end state is a last state of IHTA. *Auction(auctionID(A))*, A_1, B_3, and I_5 are the end states of their respective IHTA. Each IHTA have at least one end state. An end state is depicted as a double circle if it is an atomic state or as a double rectangle if it is a non-atomic state. All the other states in Figure 4.1 are neither start nor end states.

Section 1.2.3 motivates the requirements of concurrent processes and processes serving as abstraction of subprocesses. Atomic and non-atomic states of IHTA meet these requirements. Atomic states represent the state of elementary processes, while non-atomic states represent the state of higher processes containing elementary processes. Non-atomic states allow multiple substates to be active concurrently. The information represented by an atomic state is very basic. With an increasing number and level of concurrent substates, a non-atomic state represents information which is increasingly complex and abstract.

Statecharts [44] and Hierarchical Timed Automata (HTA) [27] support the notion of concurrent processes. However, only a fixed number of concurrent processes can be modeled by them. In the online auction use case the total number of concurrent bidder enrollment processes is, in contrast, neither known beforehand nor bound. This motivates one of IHTA's extensions (compared to Statecharts and HTA) called instantiation.

Instantiation draws a distinction between the *specification* of a non-atomic state and its *instances*. An instance i of a non-atomic state s has exactly one active state which is one of the direct substates of s. There might be zero or several instances of a non-atomic state at the same time. This resembles the relation in object oriented programming: Objects are instances of classes and there might be any number of objects instantiating a single class. In the example in Figure 4.1, each instance of the non-atomic state *BidderEnrollment(auctionID(A),enrollID(E))* has an active state which is B_0, B_1, B_2, or B_3.

Instances form a tree similarly to the tree of states. This gives rise to the notion of **subinstance** and **superinstance**. There is a connection between the tree of states and the tree of their instances: If b is an instance of the non-atomic state B, a is an instance of the non-atomic state A, then b is a subinstance of a if and only if B is a substate of A.

5.2 Transitions

The online auction use case is a system which changes its state in reaction to events. For example, three consecutive *hammerBeat* events bring the item sale into a state in which no further bids are allowed. Transitions specify the triggering conditions and the effect of these reactions.

In classical automata models such as Finite Automata [65], transitions map **source states** to **target states**. Transitions can fire when their label matches a symbol from the input word, after which the automaton is in the target state.

The classical concepts were extended considerably. For example, Timed Automata [4] use timing conditions on clocks as additional firing conditions and allow timers to be reset as reaction to the firing of a transition. Hierarchical Timed Automata (HTA) [27] introduce additional types of transitions (called pseudo transitions) which are needed for managing the state hierarchy.

IHTA transitions extend the expressiveness of transitions of Timed Automata and HTA. Transition labels of IHTA allow matching incoming events by event queries accessing the data of events, temporal constraints on the beginning and end of the occurrence time of matched events, and constraints on the number of instances.

5.2.1 Transition Role

According to their role, transitions are classified into instantiating and terminating.

An **instantiating transition** t creates an instance of each non-atomic state t goes into. Consider Figure 4.1. Transitions between states A_0 and B_0, *BidderEnrollment(auctionID(A),enrollID(E))* and B_0 create instances of the non-atomic state *BidderEnrollment(auctionID(A),enrollID(E))*. The target state of an instantiating transition is a start state.

A **terminating transition** t terminates all instances of all non-atomic states t goes out of. Consider Figure 4.1. If the transition between states B_3 and A_1 fires all instances of the state *BidderEnrollment(auctionID(A),enrollID(E))* are terminated. The source state of a terminating transition is an end state.

A transition can be either only instantiating (between *BidderEnrollment(auctionID(A),enrollID(E))* and B_0), or only terminating (between I_5 and A_1), or both instantiating and terminating (between B_3 and I_0), or neither instantiating nor terminating (between I_1 and I_2).

5.2.2 Transition Trigger

According to their trigger, transitions are classified into event and delay transitions.

Transitions triggered by events are called **event transitions**. An event transition is labeled with a non-optional positive or negative atomic *event query* and optional sets of *temporal* and *termination constraints*. The language Datalog$^{\neg,\text{time}}$ specifying transition labels is defined below.

Event queries are used for matching events. Event queries may contain *variables*. A variable X is local for the path X is declared within until the next declaration of X.

In order to differentiate between a variable declaration and a reference to the recent variable binding, the former are flagged with *. For example, X^* is a declaration of the variable X and X (without *) is a reference to the recent binding of X. The set of flagged variables and the set of unflagged variables of an event query must be disjoined.

Each variable must be declared in IHTA. A variable can be referenced in the instance i it is declared within and in the subinstances of i but not in the superinstances of i.

Each instance of a non-atomic state saves its current variable bindings. Let i be an instance with the variable bindings $VarBindings$. An event transition t fires in i if the event query q of t matches some event e of the stream with respect to $VarBindings$, i.e. $q \cdot VarBindings \cdot \sigma = e$ where σ is the set of mappings of the flagged variables of q to the respective values of e. Negated event queries may not contain variable declarations. See Definition 10 for the complete specification of the transition firing conditions.

Example: Consider the cycle involving states B_0 and B_1 in Figure 4.1. The event query *bidderData(enrollID(E),dataID(unique D*))* is used for matching events e containing the attribute *enrollID* the value of which is equal to the recent binding of the variable E. (E has been bound while firing the transition between A_0 and B_0 or the transition between *BidderEnrollment(auctionID(A),enrollID(E))* and B_0.) e must also contain the attribute $dataID$, and the variable D is bound to the respective value of e. The event queries *authentificationData(dataID(D))* and *false(dataID(D))* reference this binding of D.

An event query can be prefixed by an *event identifier* which references the event matched by the query. If q is a positive event query and j is an event identifier, $j : q$ is the declaration of j. An event identifier j is local for the path j is declared within until the next declaration of j. Each instance saves its current event identifier bindings.

The functions $b(j)$ and $e(j)$ return the beginning and the end of the occurrence time of the recent event referenced by j (compare Chapter 3). These functions are used in the temporal constraints of IHTA. Each event identifier must be declared in an instance i

before it is used in the temporal constraints of i or subinstances of i.

Since the event identifier bindings of an instance are updated *after* a transition t has fired in the instance, there would be no way for the temporal constraint c of t to identify the event which is being matched by the event query q of t. Therefore the auxiliary event identifier *this* is introduced. It references the currently matched event.

Modularization by hierarchical states, is an important design feature of IHTA. Therefore, the identifier environment's scope is local to an instance and its subinstances. In other words, an instance i cannot use event identifiers which are declared in the subinstances of i but not declared in i or a superinstance of i. To allow limited temporal constraints in transitions crossing hierarchy levels upwards, the auxiliary event identifier *last* is introduced. *last* identifies the event which was matched by the last fired event transition.

Besides, *this* and *last* are useful as "syntactic sugar", increasing readability. Thanks to *this* and *last* fewer event identifiers have to be created manually. The importance of *last* becomes especially clear when considering states with many ingoing transitions, like the state I_1 of Figure 4.1. The event transition between states I_1 and I_2 does not have to care about the various transitions which lead to the state I_1 and events matched by them. There are numerous temporal constraints in Figure 4.1 using *this* and *last*, e.g. $e(this) - e(last) < 1\text{s}$.

Please note that *this* and *last* are not defined if the current transition or the last fired transition has no event query or has a conjunction or a disjunction of multiple event queries.

Event identification in not a new feature of the event queries of IHTA. Some event query languages, e.g. XChange$^{\text{EQ}}$ [19, 33, 34], support this feature to facilitate short but expressive temporal conditions on events as needed in IHTA.

Analogously to event queries which can be prefixed by event identifiers, a state specification can be prefixed by a *state identifier*, for example *BidderEnrollment(auctionID(A),enrollID(E))* is prefixed by the identifier b. This is the declaration of the state identifier b.

State identifiers are used in termination constraints of terminating transitions. Termination constraints restrict the number of finished instances (i.e. instances in a particular end state) of a particular non-atomic state.

Example: The terminating transition between states B_3 and I_0 is labeled by the termination constraint $finishedInstances(b) \geq 2$. This means that the number of instances of the non-atomic state referenced by b (i.e. of *BidderEnrollment(auctionID(A),enrollID(E))*)

which are in the state B_3 must be at least 2 expressing that items are presented in an auction for which at least two bidders are enrolled. Otherwise the auction ends, consider the transition between states B_3 and A_1 in Figure 4.1.

Since state identifiers are used in terminating transitions crossing multiple levels of state hierarchy upwards, they are global within the run (tree of instances) they are defined in. In other words, state identifiers which are declared in an instance i can be used in i and in all sub- and superinstances of i (in contrast to variables and event identifiers which cannot be used in the superinstances of i).

Delay transitions are different from event transitions in that they are triggered by a timeout and not by an event. Therefore, delay transitions are labeled by temporal and/or termination constraints and not by an event query. *now* denotes the current time.

Example: In Figure 4.1, there is a delay-transition between the states I_0 and I_5. It expresses that the current item sale is canceled when there are no bids within 30 seconds after the item description.

5.3 Datalog$^{\neg,\text{time}}$

IHTA are defined to be fully language independent in the sense that any (complex) event query language can be used to specify the transition labels of IHTA. In this work the complex event query language, called Datalog$^{\neg,\text{time}}$, is used for this purpose. This section is devoted to the language.

Transitions of IHTA are labeled with Datalog$^{\neg,\text{time}}$ queries. If a Datalog$^{\neg,\text{time}}$ query matches its respective transition fires. A Datalog$^{\neg,\text{time}}$ query is a conjunction of (1) a positive or negative atomic event query, (2) temporal constraints, and (3) termination constraints. Consider the examples in the previous section.

Atomic event queries describe the types of matched events, their attributes and attribute values. An event query can contain variable declarations and references to the recent variable bindings. A variable flagged with * is the declaration of the variable, i.e. this variable is bound while matching events. A variable without * is the reference to the recent binding of the variable. A variable prefixed with the keyword *unique* will never be bound more than once to the same value. A positive event query prefixed with an event identifier j is the declaration of the event identifier j. j references the event matched by the query.

Temporal constraints are constraints on the occurrence time of matched events or/and on the time at which the current transition can be fired. Temporal constraints

can contain *now* which refers to the current time point, 1-ary functions $b(j)$ and $e(j)$ returning the beginning and the end of the occurrence time of the event referenced by j. Two special event identifiers *this* and *last* can be used. *this* references the currently matched event, *last* references the event matched by the last fired event transition in the considered instance. The value of *now* and the values returned by the functions are constrained by means of the binary operators $=, \neq, <, \leq, >, \geq$.

Termination constraints restrict the number of finished instances of a non-atomic state of IHTA. Termination constraints contain 1-ary functions $allInstances(s)$ and $finishedInstances(s)$ returning the number of all instances and the number of finished instances (i.e. instances in the end state which is the source state of the current transition) of the non-atomic state referenced by s. The values returned by these functions are constrained by means of the binary operators $=, \neq, <, \leq, >, \geq$.

Let l be a Datalog$^{\neg,\text{time}}$ query and t be the transition labeled by l. As explained above, if l contains an event query, t is an event transition. If l contains temporal and/or termination constraints and no event query, t is a delay transition. If l contains termination constraints, t is a terminating transition.

The grammar of Datalog$^{\neg,\text{time}}$ is defined as follows:

$TransitionLabel$	::=	$Literal?\ TempConstraint*\ TermConstraint*$
$Literal$	::=	$(EventIdentifier\ ":"\ \mid\ "not")?\ AtomicEventQuery$
$AtomicEventQuery$	::=	$Number \mid "'"?\ String\ "'"? \mid "unique"?\ Variable\ "*"?$
		$\mid String\ "("\ (AtomicEventQuery\ ",")?*\ ")"$
$TempConstraint$	::=	$Exp\ CompOp\ Exp\ (CompOp\ Exp)?$
Exp	::=	$"b("\ EventIdentifier\ ")" \mid "e("\ EventIdentifier\ ")"$
		$\mid "now" \mid Duration \mid Exp\ ArithOp\ Exp \mid "("\ Exp\ ")"$
$EventIdentifier$	::=	$Identifier \mid "this" \mid "last"$
$Duration$	::=	$(Number\ ("month"\mid"months"))?$
		$(Number\ ("week"\mid"weeks"))?$
		$(Number\ ("day"\mid"days"))?\ (Number\ ("hour"\mid"hours"))?$
		$(Number\ "min")?\ (Number\ "s")?\ (Number\ "ms")?$
$ArithOp$	::=	$"-" \mid "+" \mid "*" \mid "/" \mid "mod"$
$CompOp$	::=	$"=" \mid "\neq" \mid "<" \mid "\leq" \mid ">" \mid "\geq"$
$TermConstraint$	::=	$("finishedInstances" \mid "allInstances")$
		$"("\ StateIdentifier\ ")"\ CompOp\ Number$
$StateIdentifier$	::=	$Identifier$

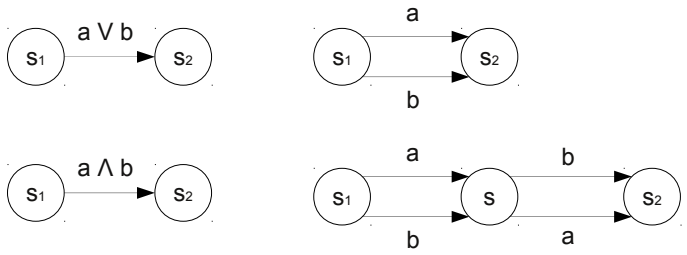

Figure 5.1: Simulation of disjunction and conjunction of two elements a and b of a transition label of IHTA.

As explained in Chapter 3, time is represented by a linearly ordered set of time points $(\mathbb{T}, \leq)$, $\mathbb{T} \in \mathbb{Q}^+$. The time interval of one second is denoted as $1s$. The time interval of one millisecond is therefore $\frac{1}{1000}s$, we write $1ms$ as a shortcut. The time interval of one minute is $60s$, we write $1min$ as a shortcut. All other possibilities to specify the duration of a time interval are computed analogously.

Note that disjunction and conjunction of multiple elements of a transition label (e.g., literals) can be simulated as shown in Figure 5.1. A disjunction of n elements leads to n additional transitions in IHTA. A conjunction of n elements causes $n!$ additional transitions and $n-1$ auxiliary states in IHTA. Nevertheless, we decided not to allow arbitrary disjunction and conjunction in Datalog$^{\neg,\text{time}}$ for the following reasons: (1) Arbitrary disjunction and conjunction can be simulated as described above. (2) IHTA are similar to the classical Nondeterministic Finite Automata [65] allowing only one symbol of the alphabet per label. (3) The event identifiers *this* and *last* are not defined if the label of the current or last transition contains a disjunction or a conjunction of multiple atomic event queries. Note that negation of an atomic event query cannot be simulated. Therefore this feature is supported by the language Datalog$^{\neg,\text{time}}$.

Of course, multiple extensions of the language are thinkable and preferable for many applications. For example, introduction of other elements into a transition label such as conditions on event data. But for this work we limit the language to the properties described above. Nevertheless the language is quite expressive and keeps IHTA readable.

Chapter 6

Semantics of IHTA

In this chapter, the semantics of Instantiating Hierarchical Timed Automata (IHTA) is formally defined. Each formal definition is preceded by an informal description and examples. Section 6.1 defines IHTA formally. Section 6.2 presents the automaton configuration and the rules for its modification.

6.1 Definition of IHTA

Let *Literals* be the set of positive or negative atomic event queries, $TempConstraints$ be a set of temporal constraints, and $TermConstraints$ be a set of termination constraints. The elements of these three sets are expressed in Datalog$^{\neg,\text{time}}$ as defined in the previous section.

Definition 1 (**Instantiating Hierarchical Timed Automaton (IHTA)**)**.** IHTA $\mathcal{I}$ is a tuple $(S, Start, End, children, T)$ where

- S is a finite set of states.
- $Start \subseteq S$ is a set of start states.
- $End \subseteq S$ is a set of end states.
- $children : S \to 2^S$ maps each state $s \in S$ to the set of its direct substates (which may be empty).[1] The function gives rise to a tree of states with root which is the root state of IHTA (Definition 3).

[1]For a set X, the notation 2^X represents the power set of X, i.e. the set of all subsets of X, formally: $2^X \stackrel{\text{def}}{=} \{x | x \subseteq X\}$.

- $T \subseteq S \times (Literals \times TempConstraints \times TermConstraints) \times S$ is the set of transitions. A transition connects two states s and s', has an optional positive or negative atomic event query q, an optional set of temporal constraints c, and an optional set of termination constraints f. We use the notation $t : s \xrightarrow{q,c,f} s' \in T$, where q, c, f can be omitted to express that they are necessarily absent. However transitions with empty labels are not allowed. A transition t without source state and with a target state $s' \in Start$ is called *enter transition*, denoted $t : \emptyset \xrightarrow{q,c} s' \in T$.

Example: The *Item Offer* represented graphically in Figure 4.1 is specified formally by the IHTA $\mathcal{I}_{ItemOffer} = (S, Start, End, children, T)$ where

$$
\begin{array}{ll}
S := & \{ItemOffer(auctionID(A), itemID(I)), I_0, I_1, I_2, I_3, I_4, I_5\} \\
Start := & \{I_0\} \\
End := & \{I_5\} \\
children := & \{ItemOffer(auctionID(A), itemID(I)) \mapsto \{I_0, I_1, I_2, I_3, I_4, I_5\}\} \cup \{x \mapsto \emptyset | x \in \{I_0, I_1, I_2, I_3, I_4, I_5\}\} \\
T := & (I_0, (bid(auctionID(A), itemID(I)), e(this) - e(last) < 30s, \emptyset), I_1), \\
 & (I_0, (\emptyset, now - e(last) \geq 30s, \emptyset), I_5), \\
 & (I_1, (bid(auctionID(A), itemID(I)), e(this) - e(last) < 30s, \emptyset), I_1), \\
 & (I_1, (hammerBeat(auctionID(A), itemID(I)), e(this) - e(last) \geq 30s, \emptyset), I_2), \\
 & (I_2, (bid(auctionID(A), itemID(I)), e(this) - e(last) < 30s, \emptyset), I_1), \\
 & (I_2, (hammerBeat(auctionID(A), itemID(I)), e(this) - e(last) \geq 30s, \emptyset), I_3), \\
 & (I_3, (bid(auctionID(A), itemID(I)), e(this) - e(last) < 30s, \emptyset), I_1), \\
 & (I_3, (hammerBeat(auctionID(A), itemID(I)), e(this) - e(last) \geq 30s, \emptyset), I_4), \\
 & (I_4, (sell(auctionID(A), itemID(I)), e(this) - e(last) < 1s, \emptyset), I_5)
\end{array}
$$

IHTA usually specify a substream of events, not the whole event stream E. Schema of IHTA describes this substream of E. Schema of IHTA is a set of atomic event queries as defined in the previous section. Only those events which match at least one of the atomic events of the schema are relevant for the respective IHTA. All other events are ignored by the IHTA.

Definition 2 (Schema of IHTA, Event Relevant for IHTA). Let *AtomicEventQueries* be the set of atomic event queries. Let $\mathcal{I}$ be IHTA. The *schema of* $\mathcal{I}$ is a set of atomic event queries, denoted $Schema(\mathcal{I}) \subseteq AtomicEventQueries$. The *default schema of* $\mathcal{I}$ is the set of atomic event queries appearing positively or negatively in at least one transition label of $\mathcal{I}$. Note that a schema of $\mathcal{I}$ always contains the default schema of $\mathcal{I}$. An event matching at least one atomic event query $q \in Schema(\mathcal{I})$ is relevant for $\mathcal{I}$. All other events are irrelevant for $\mathcal{I}$.

Example: The *Item Offer* IHTA described above have the following default schema: *Schema($\mathcal{I}_{ItemOffer}$) = { bid(auctionID(A), itemID(I)), hammerBeat(auctionID(A), itemID(I)), sell(auctionID(A), itemID(I)) }.*

Definition 3 (**Atomic State, Non-atomic State, Child State, Descendant State, Root State, Parent State, Ancestor State, Substate, Superstate**). A state $s \in S$ is

- *atomic*, if s does not contain IHTA, i.e. $children(s) = \emptyset$. s is depicted as a circle.
- *non-atomic*, if s contains IHTA itself, i.e. $children(s) \neq \emptyset$. s is depicted as a rectangle. All states $s' \in children(s)$ are *children* of s and s is the *parent* of all s', denoted $parent(s')$. *Descendants* of s are all states which are within s, formally:
  $descendants : S \to 2^S$
  $$descendants(s) = \begin{cases} \emptyset & \text{if } s \text{ is atomic} \\ children(s) \cup \bigcup\limits_{s' \in children(s)} descendants(s') & \text{otherwise} \end{cases}$$
  Descendants of s are *substates* of s, children of s are *direct substates* of s.
  A state $r \in S$ containing all other states is the *root state* of the IHTA, i.e. $r \cup descendants(r) = S$. Each IHTA has exactly one root state.
  Ancestors of s' are all states s' appears within, formally:
  $ancestors : S \to 2^S$
  $$ancestors(s') = \begin{cases} \emptyset & \text{if } s' \text{ is the root state} \\ parent(s') \cup ancestors(parent(s')) & \text{otherwise} \end{cases}$$
  Ancestors of s' are *superstates* of s', parent of s' is the *direct superstate* of s'. The root state has no parent. All other states have exactly one parent. The root state is an ancestor of all other states.

Each state of IHTA is either atomic or non-atomic. Each (atomic or non-atomic) state of IHTA is either start or end or neither start nor end state. An end state s is depicted as a double circle if s is atomic or as a double rectangle if s is non-atomic. Consider examples in Chapter 5.

Definition 4 (**Event Transition, Delay Transition**). A transition $t : s \xrightarrow{q,c,f} s' \in T$ is

- *event* if $q \neq \emptyset$.
- *delay* if $q = \emptyset$.

Each transition is either event or delay. Consider examples in Chapter 5.

Definition 5 (**Instantiating Transition, Terminating Transition**). A transition $t : s \xrightarrow{q,c,f} s' \in T$ is

- *instantiating* if t goes into a (set of nested) non-atomic state(s) and $s' \in Start$.

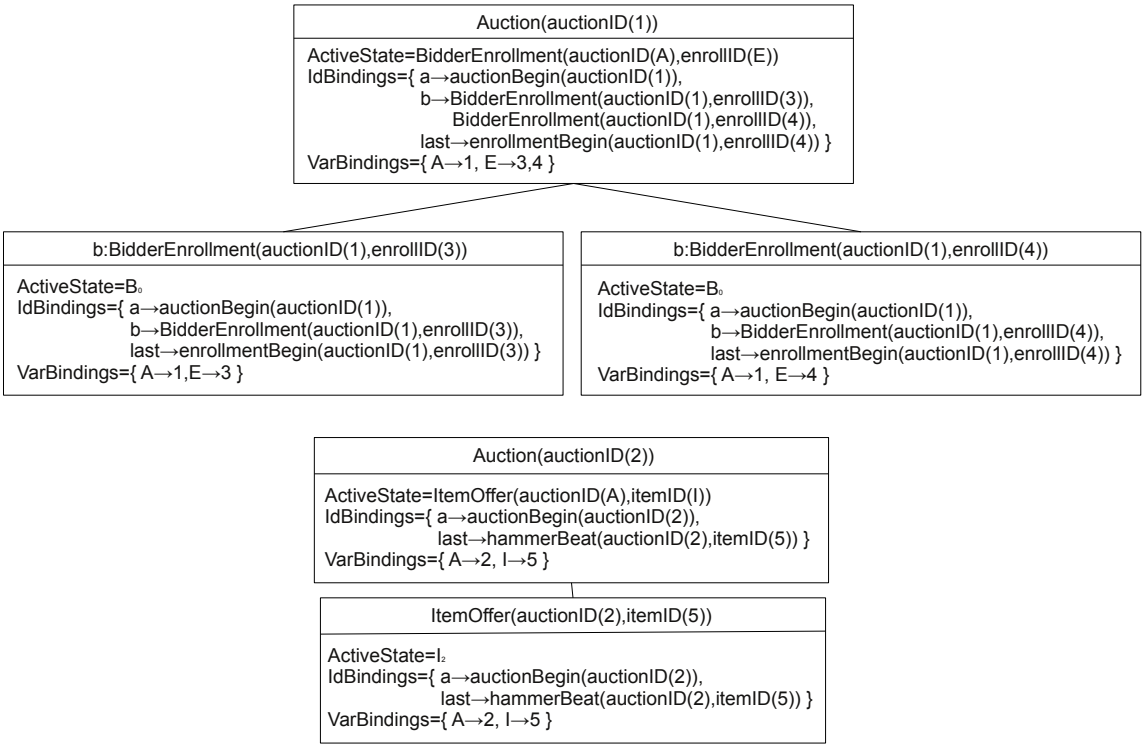

Figure 6.1: Graphical representation of two runs of IHTA in Figure 4.1. An instance is represented by a box. The tree of instances is called run.

- *terminating* if t goes out of a (set of nested) non-atomic state(s) and $s \in End$.

All properties of transitions are independent from each other and can be freely combined in the sense that an event or a delay transition can be (a) neither instantiating nor terminating, (b) instantiating but not terminating, (c) terminating but not instantiating or (d) both instantiating and terminating. Consider examples in Chapter 5 and Figure 6.3.

6.2 Automaton Configuration

Having specified the basic notions IHTA are comprised of, we will now present the automaton configuration, a structure which represents the entire state during the runs of an automaton. It includes all sets of nondeterministic runs, all instances of non-atomic states in each run and, for each instance, its active state, variable bindings, and event and state identifier bindings.

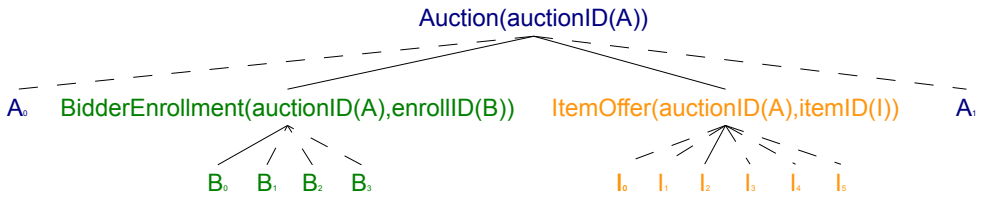

Figure 6.2: The run modeling approach of Hierarchical Timed Automata [27].

6.2.1 Instance of a Non-Atomic State

Instantiation allows to represent an arbitrary number of concurrent processes. Consider Figure 6.1. *Auction(auctionID(1))* and *Auction(auctionID(2))* are instances of the state *Auction(auctionID(A))* (Definition 6). These instances represent two auctions taking place at the same time independently from each other. Both auctions follow the predefined workflow specified by the IHTA within the state *Auction(auctionID(A))*. The active states of the instances are *BidderEnrollment(auctionID(1),enrollID(E))* and *ItemOffer(auctionID(2),itemID(I))* respectively (Definition 8). Each of the instances has its own variable bindings and identifier bindings. Each of the concurrent processes can give rise to new concurrent processes. So for example, during the auction with identifier 1 two bidder enrollment processes take place at the same time, i.e. the instance *Auction(auctionID(1))* has two subinstances instantiating the nonatomic state *BidderEnrollment(auctioID(A),enrollID(E))*. Instances of non-atomic states form a tree called run (Definition 12). Figure 6.1 shows two runs of the IHTA in Figure 4.1. A run cannot be modeled as a partial tree of the state tree because there can be more than one instances with the same active state, e.g. *BidderEnrollment(auctionID(1),enrollID(3)* and *BidderEnrollment(auctionID(1),enrollID(4)*.

Hierarchical Timed Automata (HTA) [27] do not have the notion of instantiation. A run of HTA is presented as a partial tree of the state tree. Figure 6.2 shows an example of this approach. Edges which are bold represent a tree of states which are currently active. The approach cannot represent a state being active more than once in a run. Therefore, only a fixed number of concurrent processes can be modeled. This does not meet our requirements of modeling an arbitrary number of concurrent processes. IHTA handle this by *instantiation*. A run is built top-down during run-time in the following way. When an enabled transition t (Definition 10) enters a non-atomic state n, an instance i of n is created. If n is the root state, i has no parent. Otherwise, the parent of i is the instance firing t. i inherits the variable bindings and the identifier bindings from its parent, but only those which were not yielded during creating the siblings of i.

Every instance has a set of local variable bindings and identifier bindings. These are

updated when a transition fires in the instance. If a transition fires in an instance, the active state of the instance changes. This procedure is called *transformation*.

Sibling instances represent concurrent processes. Subinstances of an instance can *terminate*.

These three steps, i.e. instantiation, transformation, and termination, are executed every time a transition fires (Definition 16).

Now, let us formally define the notions which were informally explained above. Let $\mathcal{I}$ be IHTA. Let $IdentifierBindings$ be the set of event (or state) identifier bindings and $VariableBindings$ be the set of variable bindings.

Definition 6 (**Instance of a Non-Atomic State**)**.** An instance i of a non-atomic state $n \in S$ of $\mathcal{I}$ is a tuple $(n \cdot VarBindings, ActiveState, IdBindings, VarBindings)$ where

- $n \cdot VarBindings$ is the name of i which is the result of the application of $VarBindings$ (as defined below) to n.
- $ActiveState \in children(n)$ is the active state of i (Definition 8).
- $IdBindings \in IdentifierBindings$ is the set of identifier bindings of i.
- $VarBindings \in VariableBindings$ is the set of variable bindings of i.

Definition 7 (**Schema of an Instance, Event Relevant for an Instance**)**.** Let $n \in S$ be a nonatomic state of IHTA with the schema $Schema(n)$. Let i be an instance of n with the variable bindings $VarBindings$. The schema of i is the schema of n in which the variable bindings of i are propagated, formally $Schema(i) = \{q \cdot VarBindings \mid q \in Schema(n)\}$. An event is relevant for i if it matches at least one atomic event query of the schema of i. Otherwise, an event is irrelevant for i.

Example: If the schema of the IHTA *Item Offer* $\mathcal{I}_{ItemOffer}$ in Figure 4.1 is *Schema($\mathcal{I}_{ItemOffer}$) = { bid(auctionID(A), itemID(I)), hammerBeat(auctionID(A), itemID(I)), sell(auctionID(A), itemID(I)) }* then the schema of its instance $i_{ItemOffer}$ in Figure 6.1 is *Schema($i_{ItemOffer}$) = { bid(auctionID(2), itemID(5)), hammerBeat(auctionID(2), itemID(5)), sell(auctionID(2), itemID(5)) }*.

Consider Figure 6.3.

Definition 8 (**Active State of an Instance**)**.** Let i be an instance of a non-atomic state $n \in S$. If $t : s \xrightarrow{q,c,f} s' \in T$ is an enabled transition in i (Definition 10) then the

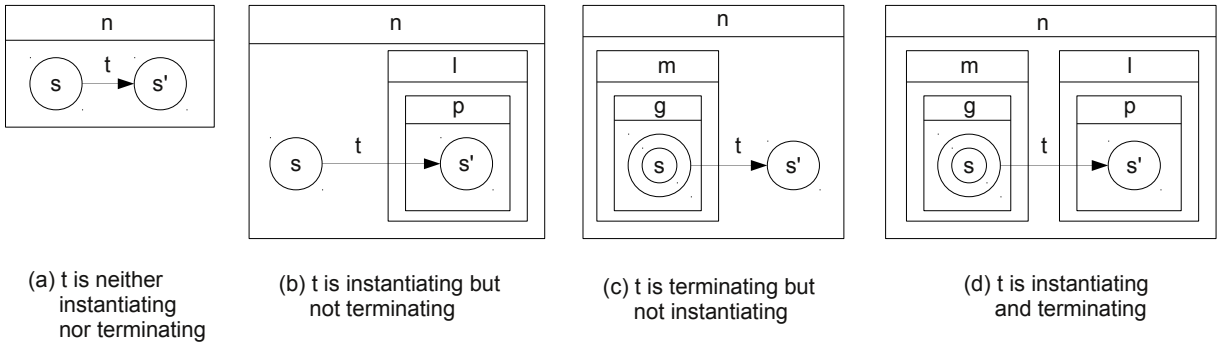

Figure 6.3: Four kinds of transitions illustrated by the (incomplete) IHTA

active state of i, denoted $activeState(i)$, is defined as follows:

$$activeState(i) = \begin{cases} l \mid l \in children(n) \wedge l \in ancestors(s') & \text{if } t \text{ is instantiating} \\ s' & \text{else} \end{cases}$$

Everytime an enter transition of IHTA $\mathcal{I}$ fires a new run of $\mathcal{I}$ is created. An enter transition $t : \emptyset \xrightarrow{q,c} s' \in T$ of $\mathcal{I}$ is performed if the following condition is satisfied: If the event query q of t is positive, there must be an event in the stream which is matched by q such that the optional temporal constraints c of t are satisfied. If q is negative, i.e. $q = \neg q'$, q matches the event stream if the stream does not contain an event matched by q' such that the non-optional temporal constraints c of t are satisfied.

Definition 9 (**Enabled Enter Transition**)**.** Let E be an event stream. Let $\mathcal{I}$ be IHTA. An enter transition $t : \emptyset \xrightarrow{q,c} s' \in T$ of $\mathcal{I}$ is enabled if:

- if $c = \emptyset$ then q is positive and $\exists a^d \in E, \exists \sigma$ such that $q\sigma = a$.
- if $c \neq \emptyset$ then let $d' \in \mathbb{TI}$ be the time interval during which c is satisfied
  - if q is positive then $\exists a^d \in E, \exists \sigma$ such that $q\sigma = a$ and $d \sqsubseteq d'$[2] .
  - if q is negative, i.e. $q = \neg q'$, then $\nexists a^d \in E$ for which $\exists \sigma$ such that $q\sigma = a$ and $d \sqsubseteq d'$.

An instance i of a non-atomic state $n \in S$ executes a transition $t : s \xrightarrow{q,c,f} s' \in T$ if the following conditions are satisfied:

- The instance i is in the right state which depends on the kind of the transition t. If t is:

[2] $d \sqsubseteq d'$ is the shortcut for $b(d') \leq b(d)$ and $e(d) \leq e(d')$. In other words the time interval d' comprises the time interval d.

  - neither instantiating nor terminating (Case (a) in Figure 6.3), $s, s' \in children(n)$ and s is the active state of i.
  - instantiating but not terminating (Case (b) in Figure 6.3), $s \in children(n), s' \in descendants(n)$ and s is the active state of i.
  - terminating but not instantiating (Case (c) in Figure 6.3), $s \in descendants(n), s' \in children(n)$ and m is the active state of i where $m \in children(n), m \in ancestors(s)$.
  - instantiating and terminating (Case (d) in Figure 6.3), $s, s' \in descendants(n)$ and m is the active state of i where $m \in children(n), m \in ancestors(s)$ and there is no state k such that $s, s' \in descendants(k), k \in descendants(n)$ (otherwise an instance of k performs t).

- The components of the label of t, namely the event query q, the temporal constraints c, and the termination constraints f are satisfied.
  - If t is an event transition and the event query q of t is positive, there must be an event in the stream which is matched by q such that the optional temporal constraints c and the optional termination constraints f of t are satisfied with respect to the variable bindings and identifier bindings of i.

    If q is negative, i.e. $q = \neg q'$, q matches the event stream if the stream does not contain an event matched by q' such that the temporal constraints c and the termination constraints f of t are satisfied with respect to the variable bindings and identifier bindings of i. Please note that in this case either temporal or termination constraints are non-optional.
  - The temporal constraints c of t are satisfied with respect to the identifier bindings of i.
  - If t is terminating, the termination constraints f of t are satisfied with respect to the identifier bindings of i.

Definition 10 (**Transition Enabled in an Instance**)**.** Let i be an instance of a non-atomic state $n \in S$, let $VarBindings$ be the variable bindings of i and $IdBindings$ be the identifier bindings of i. Let E be an event stream. A transition $t : s \xrightarrow{q,c,f} s' \in T$ is enabled in i if:

1. $activeState(i) = \begin{cases} m \mid m \in children(n) \wedge m \in ancestors(s) \wedge \\ \quad \nexists k \mid s, s' \in descendants(k) \wedge \\ \quad\quad k \in descendants(n) & \text{if } t \text{ is terminating} \\ s & \text{else} \end{cases}$

2. if t is an event transition then

   - if $c = \emptyset$ and $f = \emptyset$ then q is positive and $\exists a^d \in E, \exists \sigma$ such that $q \cdot VarBindings \cdot \sigma = a$.
   - if $c \neq \emptyset$ or $f \neq \emptyset$ then let $d' \in \mathbb{TI}$ be the time interval during which $c \cdot IdBindings$ and $f \cdot IdBindings$ are satisfied
     - if q is positive then $\exists a^d \in E, \exists \sigma$ such that $q \cdot VarBindings \cdot \sigma = a$ and $d \sqsubseteq d'$.
     - if q is negative, i.e. $q = \neg q'$, then $\nexists a^d \in E$ for which $\exists \sigma$ such that $q' \cdot VarBindings \cdot \sigma = a$ and $d \sqsubseteq d'$.

3. if $c \neq \emptyset$ then $c \cdot IdBindings$ is satisfied.

4. if t is terminating then $f \cdot IdBindings$ is satisfied.

Just analogously to Definition 3 for states, the following sets are defined for an instance:

Definition 11 (**Child Instance, Descendant Instance, Parent Instance, Ancestor Instance, Subinstance, Superinstance**)**.** Let $n, n' \in S$ such that $n' \in children(n)$. Let i be an instance of n and i' be an instance of n' such that i' was created by an instantiating transition in i or both i and i' were created by an instantiating transition in an ancestor instance of i. Then i' is a *child* of i, denoted $i' \in children(i)$, and i is *parent* of i', denoted $parent(i') = i$. Let I be the set of instances. *Descendants* of i are the following:

$descendants : I \to 2^I$

$$descendants(i) = \begin{cases} \emptyset & \text{if the active state of } i \text{ is atomic} \\ children(i) \cup \bigcup_{i' \in children(i)} descendants(i') & \text{otherwise} \end{cases}$$

Descendants of i are *subinstances* of i, children of i are *direct subinstances* of i.

Ancestors of i' are the following:

$ancestors : I \to 2^I$

$$ancestors(i') = \begin{cases} \emptyset & \text{if } i' \text{ is an instance of the root state} \\ parent(i') \cup ancestors(parent(i')) & \text{otherwise} \end{cases}$$

Ancestors of i' are *superinstances* of i', parent of i' is the *direct superinstance* of i'. An instance of the root state has no parent. All other instances have exactly one parent. An instance of the root state is an ancestor of all other instances.

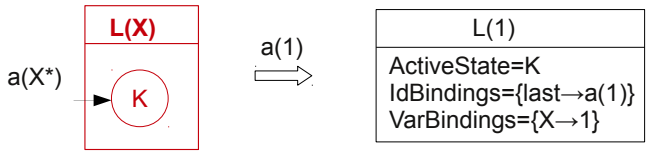

Figure 6.4: The (incomplete) IHTA and its initial run. The enter transition instantiates only the root state.

6.2.2 Run

A run r saves all current instances in r as well as their name, active state, identifier bindings and variable bindings.

Definition 12 (Run). Let $\mathcal{I}$ be IHTA. Let I be the set of instances and $Names$ be the set of instance names. A run r of $\mathcal{I}$ is a tuple ($children$, $name$, $activeState$, $identifierBindings$, $variableBindings$) where

- $children : I \to 2^I$ maps an instance $i \in I$ to its children. $children(i)$ gives rise to a tree of instances the root of which is an instance of the root state.
- $name : I \to Names$ maps an instance $i \in I$ to its name which is the result of the application of variable bindings of i to the state $n \in S$ instantiated by i, i.e. $name(i) = n \cdot variableBindings(i)$.
- $activeState : I \to S$ maps an instance to its active state.
- $identifierBindings : I \to IdentifierBindings$ maps an instance to its identifier bindings.
- $variableBindings : I \to VariableBindings$ maps an instance to its variable bindings.

Definition 13 (Event Relevant for a Run). Let r be a run of IHTA and I_r be the set of instances in r. An event is relevant for r if it is relevant for at least one instance in I_r. Otherwise, an event is not relevant for r.

Definition 14 (Initial Run). Let $\mathcal{I}$ be IHTA. Let $t : \emptyset \xrightarrow{q,c} s' \in T$ be an enabled enter transition of $\mathcal{I}$. Let $root \in S$ be the root state of $\mathcal{I}$. Initially there is one run r. It contains an instance i of $root$ without subinstances if t instantiates only $root$. If t instantiates $root$ and substates l of $root$, r contains an instance i of $root$ and an instance i' of each l which

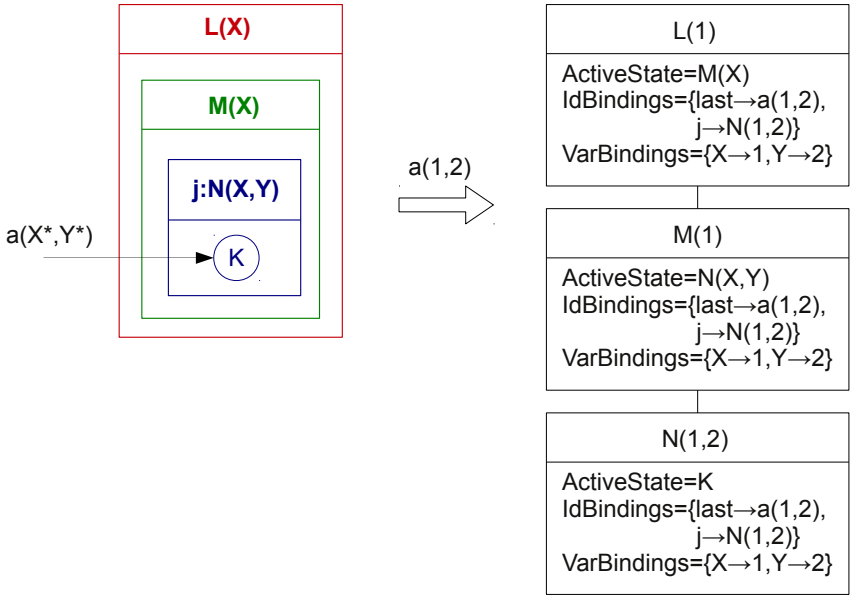

Figure 6.5: The (incomplete) IHTA and its initial run. The enter transition instantiates the root state and its two substates.

are subinstances of i. $r := Initialize_t()$ which is defined as follows:

$$
\begin{aligned}
& i \text{ is a new instance of } root \\
children &:= i \mapsto \emptyset \\
activeState &:= \begin{cases} i \mapsto s' & \text{if } t \text{ instantiates only } root \\ i \mapsto l \mid l \in children(root) \wedge & \text{if } t \text{ instantiates } root \text{ and} \\ \qquad l \in ancestors(s') & \text{substates of } root \end{cases} \\
identifierBindings &:= i \mapsto \{last \mapsto e \cup id \mapsto e\} \\
variableBindings &:= i \mapsto \sigma \\
name &:= i \mapsto root \cdot \sigma \\
\text{return } & Inst(i, s', children, name, activeState, identifierBindings, variableBindings)
\end{aligned}
$$

where
id is the event identifier, i.e. $q = id : q'$,
e is the event matched by q, and
σ is the unifier of q and e, i.e. $q\sigma = e$.
In this case we say that **t is enabled by e**.
The function *Inst(i, s', children, name, activeState, identifierBindings, variableBindings)* is given in Definition 16.

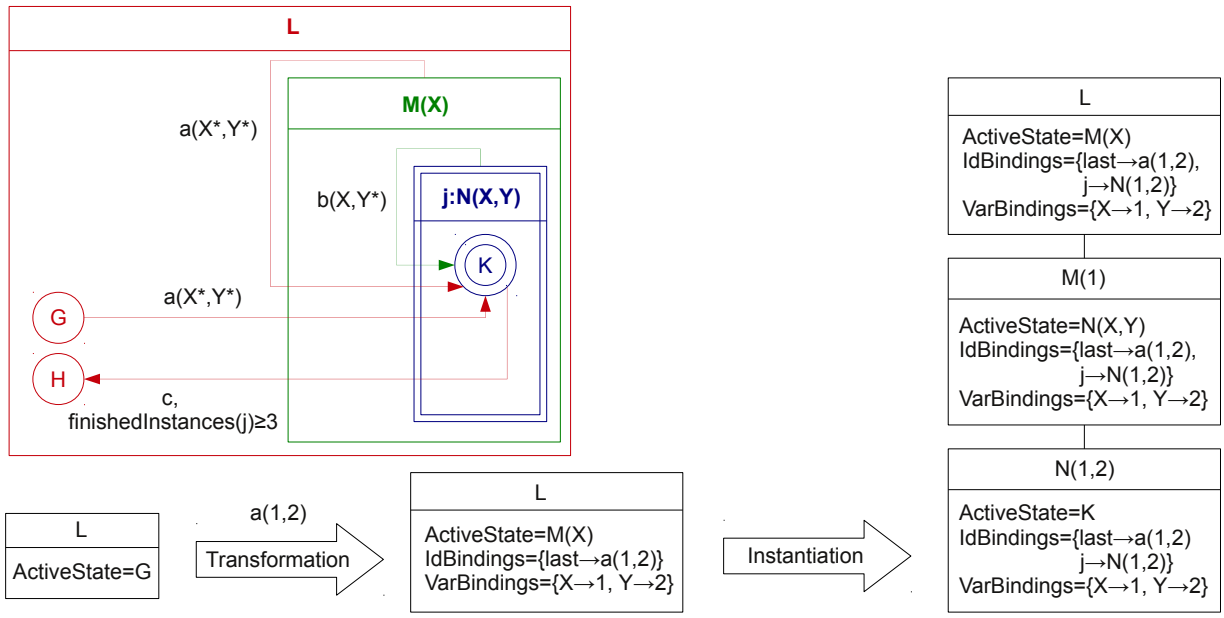

Figure 6.6: The (incomplete) IHTA and its run modification involving transformation and instantiation.

Consider the (incomplete) IHTA and their respective initial runs in Figure 6.4 and Figure 6.5.

Run Modification

Let r be a run of IHTA $\mathcal{I}$ and I_r be the set of instances of r. Let $t : s \xrightarrow{q,c,f} s' \in T$ be a transition enabled in an instance $i \in I_r$ of a nonatomic state $n \in S$. When t fires in i, r is modified as follows:

1. Transformation: If s' is a child of n then s' becomes the active state of i. Otherwise, the active state of i is the ancestor of s' which is a child of n. (Compare Definition 8.) If t is an event transition, the variable bindings and identifier bindings of i are updated. The variable bindings of i are updated using the unifier of the event query q of t and the current event e. If q is labeled by an event identifier id, the identifier bindings of i are updated such that id is mapped to e. The identifier bindings of all ancestors of i are also updated: The event identifier $last$ references e.

Example: Consider Figure 6.6. Assume the event $a(1, 2)$ arrives. The transition $t : G \xrightarrow{a(X^*,Y^*)} K$ is enabled in the instance i of L (compare Definition 10). $K \notin children(L)$ and the active state of i is $M(X)$ because $M(X) \in ancestors(K)$ and $M(X) \in children(L)$. The variable bindings and the identifier bindings of i are updated as shown in Figure 6.6.

2. Instantiation: If t is instantiating, then for each state t goes into (more exactly,

for each ancestor of s' which is a descendant of n) a new instance i' is created. All these instances are in a parent-child relation as defined in Definition 11. Each new instance i' inherits variable bindings and identifier bindings from its parent but only those which were not yielded while creating siblings of i'. The active state of each new instance i' is either s' or the state which is instantiated by the children of i'. If a state k instantiated by t carries a state identifier id' the identifier bindings of the instance i' of k are updated so that id' is mapped to the name of i'. This state identifier binding is provided to all ancestors of i'.

Example: Consider Figure 6.6 again. The enabled transition $t : G \xrightarrow{a(X^*,Y^*)} K$ in i is instantiating. An instance for each state $M(X)$ and $N(X,Y)$ is created because $M(X), N(X,Y) \in \mathit{ancestors}(K)$ and $M(X), N(X,Y) \in \mathit{descendants}(L)$. $M(1)$ is the instance of $M(X)$, $M(1) \in \mathit{children}(i)$, and $N(1,2)$ is the instance of $N(X,Y)$, $N(1,2) \in \mathit{children}(M(1))$. The active state of $M(1)$ is $N(X,Y)$ and the active state of $N(1,2)$ is K. Each instance inherits the variable bindings and the identifier bindings from its parent instance.

Modeling Concurrent Processes

A special form of instantiating transitions are transitions with a non-atomic source state s and a start target state s' where s' is a descendant of s. Figure 6.7 shows the result of the execution of two instantiating transitions of this form. In contrast to the instantiation shown in Figure 6.6, newly created subinstances are siblings of already existing subinstances. Since each of these subinstances processes transitions independently from other subinstances this models multiple concurrent processes.

To relate an event to an instance, event queries refer to the variable bindings of an instance. Variables used for this relation are usually declared in the event queries of instantiating transitions (see Chapter 5). This requires special care from the user in the following two respects: (1) Variables used for *uniquely* relating events to instances must be keys. (2) It is not prevented that variables are accidentally shadowed during the lifetime of an instance. For example, the variable Y in the instance $M(1)$ is shadowed after the first instantiation in Figure 6.7. But its subinstance $N(1,2)$ still has the old binding of Y.

3. Termination: If t is terminating, all finished subinstances i' of the instance i with respect to the end state s are terminated. As defined in Definition 15, an instance i' is finished with respect to the end state s if either:

- the active state of i' is s or
- the active state of i' is a non-atomic state n (i.e. i' has subinstances i'' instantiating

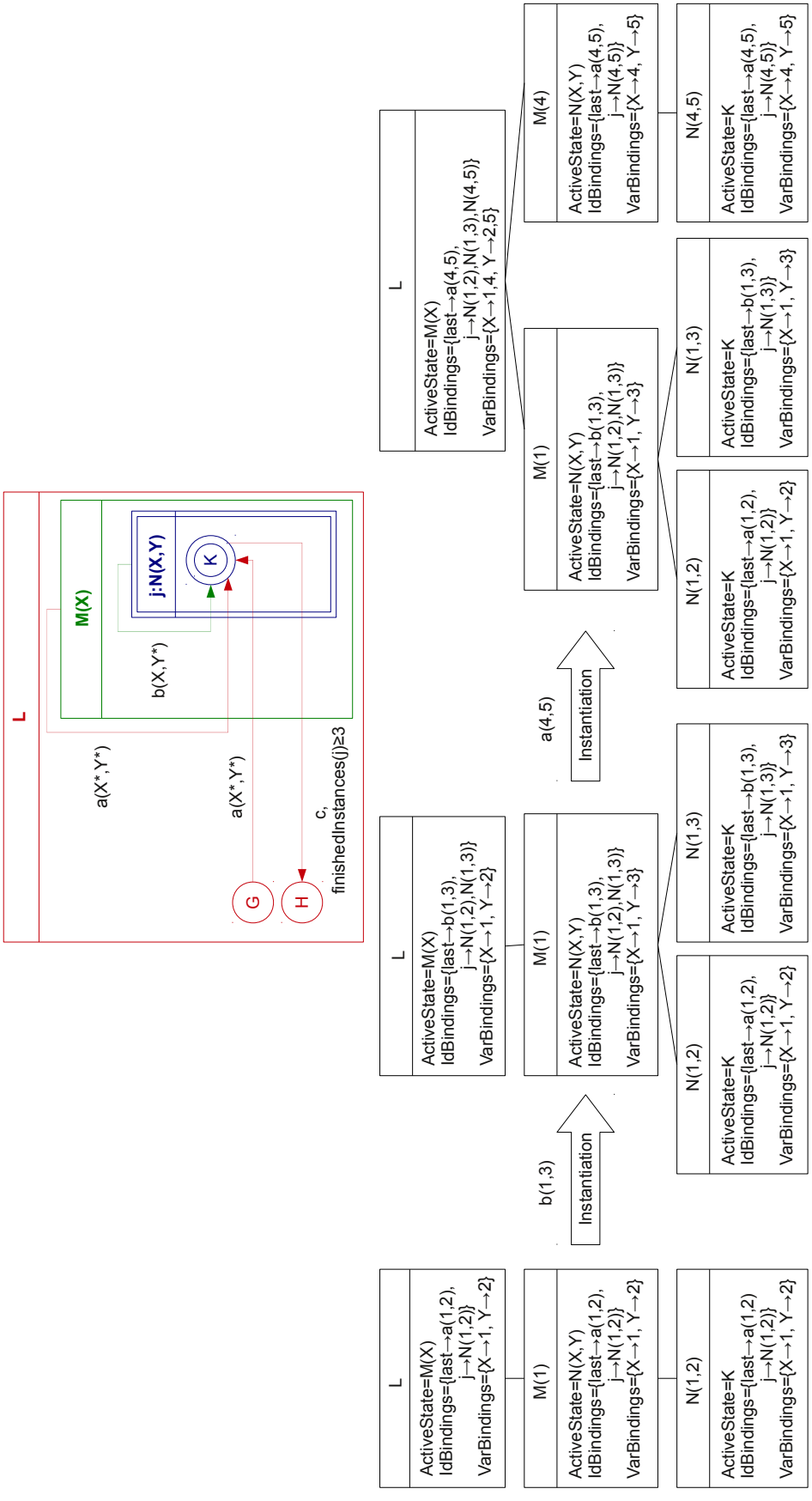

Figure 6.7: Modeling concurrent processes

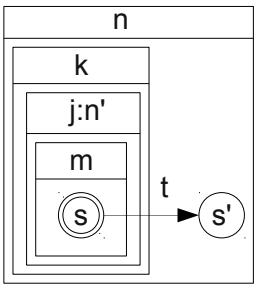

Figure 6.8: State identifier j in termination constraints of t illustrated by the (incomplete) IHTA

n) and all instances i'' are finished with respect to s.

Example: Consider the resulting run in Figure 6.7. All instances of the non-atomic state $N(X, Y)$ are finished with respect to the end state K according to the first condition above. Both instances of the non-atomic state $M(X)$ are finished with respect to the end state K according to the second condition above.

Definition 15 (Finished Instance with respect to an End State). Let i be an instance of a non-atomic state $n \in S$, let $s \in descendants(n)$ be an atomic end state. i is called finished with respect s if either

- $activeState(i) = s$
  or
- $activeState(i)$ is non-atomic and all instances of $activeState(i)$ which are subinstances of i are finished with respect to s.

Termination Constraints

In the auction use case, for an item to be presented in an auction there must be at least two bidder enrollments. Therefore, the event transition between states B_3 and I_0 in Figure 4.1 is only executed if at least two instances of the non-atomic state *b:BidderEnrollment(auctionID(A),enrollID(E))* are in the state B_3. Such conditions on the number of instances are expressed by termination constraints of terminating transitions. Termination constraints are formulas of arithmetic expressions comparing the value returned by the functions $allInstances$ or $finishedInstances$ (defined below) with a natural number by means of the binary operators $=, \neq, <, \leq, >, \geq$.

Let $t : s \xrightarrow{q,c,f} s' \in T$ be a terminating transition in an instance i of a non-atomic state $n \in S$ in a run r. Let $j : n' \in S$ be a non-atomic state referenced by the state identifier

j such that $j : n' \in descendants(n)$ and $j : n' \in ancestors(s)$. Consider Figure 6.8. Two functions $allInstances$ and $finishedInstances$ can be used in the termination constraint f of t. They are defined as follows:

- The function $allInstances : Identifiers \to \mathbb{N}_0$ maps j to the number of instances of n' regardless their active states in r.
- The function $finishedInstances : Identifiers \to \mathbb{N}_0$ maps j to the number of finished instances of n' with respect to s in r.

If t does not have an explicit termination constraint then it carries a **default termination constraint** $allInstances(l) = finishedInstances(l)$ where l references all states x for which $x \in descendants(n)$ and $x \in ancestors(s)$ holds (these are k, n' and m in Figure 6.8). Default termination constraint expresses that all instances of all substates of n in r must be finished with respect to s. If a terminating transition is labeled by an explicit termination constraint f, the default termination constraint is replaced by f.

Example: The termination constraint of the transition between states B_3 and I_0 in Figure 4.1 is $finishedInstances(b) \geq 2$. Therefore the default termination constraint $allInstances(b) = finishedInstances(b)$ is replaced by the custom termination constraint, and the transition can fire even if not all instances of the non-atomic state *b:BidderEnrollment(auctionID(A),enrollID(E))* are finished with respect to B_3 but at least two.

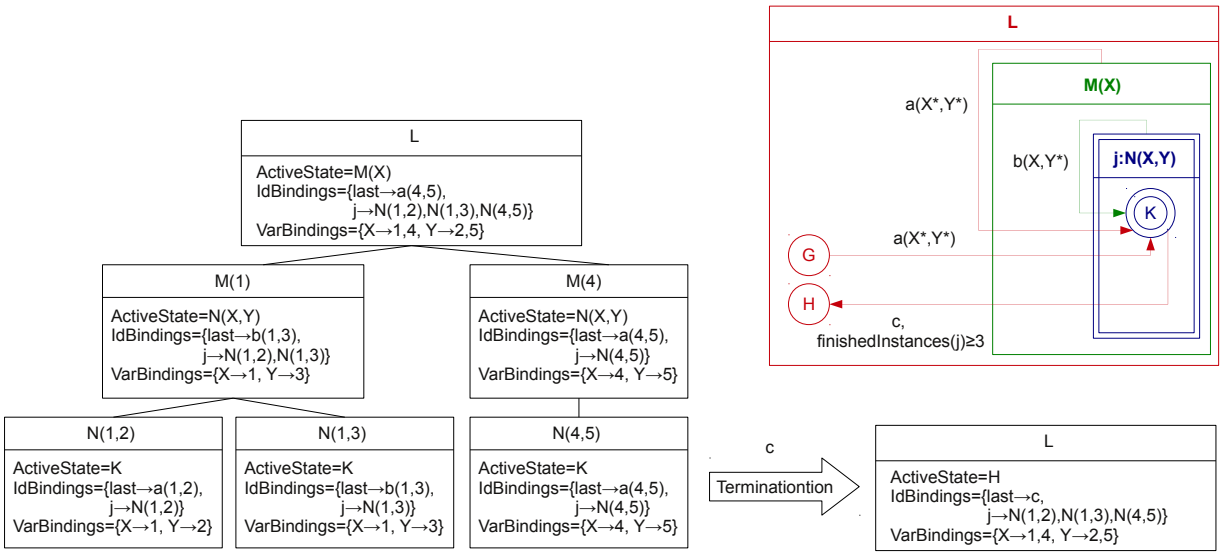

Figure 6.9: The (incomplete) IHTA and its run modification involving termination

Example: Assume the event c arrives. Then the terminating transition between states K and H in Figure 6.9 is enabled, compare Definition 10. (If j referenced the non-atomic

state $M(X)$ the termination constraint would not be satisfied and the transition would not be enabled.) All subinstances of the instance i of L which are finished with respect to K terminate. The active state of i is now H.

Definition 16 formally specifies the run modification including tree steps namely termination, transformation and instantiation which were informally described and illustrated by examples above.

Definition 16 (Deterministic Run Modification). Let $r = (children,\ name,\ activeState,\ identifierBindings,\ variableBindings)$ be a run of IHTA. Let I_r be the set of instances in r and $i \in I_r$ instantiating $n \in S$. Let $t : s \xrightarrow{q,c,f} s' \in T$ be the only enabled transition in i. Performing t in i is executed by the deterministic modification function

$$\mathcal{D}_{i,t}(children, name, activeState, identifierBindings, variableBindings) \stackrel{def}{=} (children_3, name_3, activeState_3, identifierBindings_3, variableBindings_3)$$

which takes r as an input and returns a modified run ($children_3$, $name_3$, $activeState_3$, $identifierBindings_3$, $variableBindings_3$) as an output. The execution of $\mathcal{D}_{i,t}$ involves three steps:

1. **Termination**
   If t is terminating the subinstances of i are terminated. For this effect, an intermediate run ($children_1$, $name_1$, $activeState_1$, $identifierBindings_1$, $variableBindings_1$) is created:

$$children_1 := \begin{cases} children[i \mapsto \emptyset] & \text{if } t \text{ is terminating} \\ children & \text{else} \end{cases}$$

$$name_1 := name$$
$$activeState_1 := activeState$$
$$identifierBindings_1 := identifierBindings$$
$$variableBindings_1 := variableBindings$$

2. **Transformation**
   The active state of i is updated. If t is an event transitions, the event identifiers and variable bindings of i are updated. An intermediate run ($children_2$, $name_2$,

$activeState_2$, $identifierBindings_2$, $variableBindings_2$) is created:

$$children_2 := children_1$$

$$name_2 := name_1$$

$$activeState_2 := \begin{cases} activeState_1[i \mapsto l \mid l \in children(n) \wedge & \\ \quad l \in ancestors(s')] & \text{if } t \text{ is instantiating} \\ activeState_1[i \mapsto s'] & \text{else} \end{cases}$$

$$identifierBindings_2 := \begin{cases} identifierBindings_1[i \mapsto & \\ \quad \{identifierBindings_1(i) \cup & \\ \quad last \mapsto e \ \cup \ id \mapsto e\}, & \\ \quad \forall a \in ancestors(i) : a \mapsto & \\ \quad \{identifierBindings_1(a) \cup & \\ \quad last \mapsto e\}] & \text{if } t \text{ is an event transition} \\ identifierBindings_1 & \text{else} \end{cases}$$

$$variableBindings_2 := \begin{cases} variableBindings_1[i \mapsto & \\ \quad \{variableBindings_1(i) \cup \sigma\}] & \text{if } t \text{ is an event transition} \\ variableBindings_1 & \text{else} \end{cases}$$

where
id is the event identifier, i.e. $q = id : q'$,
e is the event matched by $q \cdot variableBindings_1(i)$, and
σ is the unifier of $q \cdot variableBindings_1(i)$ and e, i.e. $q \cdot variableBindings_1(i) \cdot \sigma = e$.
In this case we say that **t is enabled by e**.

3. **Instantiation**
   If t is instantiating, then for each state t goes into a new instance is created. The resulting run is ($children_3$, $name_3$, $activeState_3$, $identifierBindings_3$, $variableBindings_3$) := $Inst(i, s', children_2, name_2, activeState_2, identifierBindings_2, variableBindings_2)$ which is defined as follows:

$$Inst(i, s', children, name, activeState, identifierBindings, variableBindings) \stackrel{\text{def}}{=}$$

$$\begin{cases} Inst(i', s', children', name', activeState', \\ \quad identifierBindings', variableBindings'), \text{ where} \\ \quad i' \text{ is a new instance of } activeState(i) \\ \quad children' := children[i \mapsto \{children(i) \cup i'\}] \\ \quad activeState' := activeState[i' \mapsto l \mid \\ \quad\quad l \in children(activeState(i)) \wedge \\ \quad\quad (l \in ancestors(s') \vee l = s')] \\ \quad variableBindings' := variableBindings[i' \mapsto \\ \quad\quad \{variableBindings(i) - varBindings(i)\}] \\ \quad name' := name[i' \mapsto activeState(i) \cdot variableBindings'(i')] \\ \quad identifierBindings' := identifierBindings[i' \mapsto \\ \quad\quad \{(identifierBindings(i) \cup id' \mapsto name') \\ \quad\quad -idBindings(i)\}, \\ \quad\quad \forall a \in ancestors(i) : a \mapsto \\ \quad\quad \{identifierBindings(a) \cup id' \mapsto name'\}] & \text{if } activeState(i) \neq s' \\ \\ (children, name, activeState, identifierBindings, \\ \quad\quad variableBindings) & \text{else} \end{cases}$$

where
id' is the state identifier of the active state of i instantiated by i',
$idBindings(i)$ and $varBindings(i)$ be the identifier bindings and the variable bindings of i yielded while creating siblings of i'. These bindings are irrelevant for i'.

6.2.3 Set of Nondeterministic Runs

Definition 16 specifies run modification if there is only one enabled transition in an instance, i.e. deterministic run modification. However, IHTA are non-deterministic in general, i.e. there can be multiple transitions enabled in an instance at the same time. It is not known beforehand which of these transitions leads to the end state. Therefore, IHTA support don't know (or disjunctive) nondeterminism (consider Section 1.2.5). Our understanding of nondeterminism is based on Nondeterministic Finite Automata NFA [65].

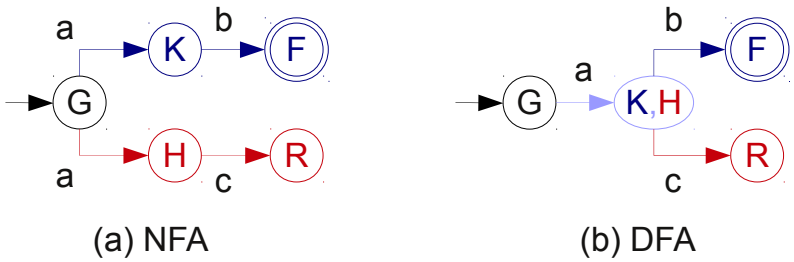

Figure 6.10: Powerset construction

When multiple transitions of an NFA can be executed, *any* of them is chosen. As long as there is *at least one* run of the NFA which leads to an end state, the input word is accepted.

Powerset construction method [65], i.e. translation of NFA into Deterministic Finite Automata (DFA), is a solution to the ambiguous choice between multiple enabled transitions. As motivated in Section 1.2.5, powerset construction allows event stream verification on-the-fly, i.e. there is no need to save past events in order to test other choices.

Example: Figure 6.10 shows an exemplary NFA and its respective DFA without the states which are unreachable from the start state. Assume the active state is G and the event a arrives. Two transitions of the NFA are enabled, the one between states G and K and the one between states G and H. Both runs are followed. They are depicted in blue and red in Figure 6.10. The active state of the blue run is K and the active state of the red run is H. This behavior is simulated by merging the states K and H in the DFA. Assume the event b arrives. The red run of the NFA cannot accept it and is therefore dropped. The blue run accepts it and is therefore stil valid. (If the blue run did not accept b, the word would not be in the language specified by the NFA.) The active state is F and the word is accepted. Note that the same behavior is simulated by the DFA.

As the example shows, powerset construction method relies on the fact that the automaton is finite, i.e. has a finite number of states. There is no way of using it for IHTA because the number of instances of states is unbounded in each run. However, powerset construction method can be reproduced by keeping track of all possible nondeterministic runs of IHTA (which are called *a set of nondeterministic runs* in the following). When an instance in a run r is in a state with k enabled transitions, r is branched into k runs $r_1, \ldots, r_k$. Each of $r_1, \ldots, r_k$ performs one enabled transition. $r_1, \ldots, r_k$ belong to the same set of nondeterministic runs as r. All sets of nondeterministic runs are accumulated by *automaton configuration* and modified dynamically. When an event e occurs, runs of the respective set of nondeterministic runs which do not have an instance with event transitions which are triggered by e are deleted. If a set of nondeterministic runs becomes

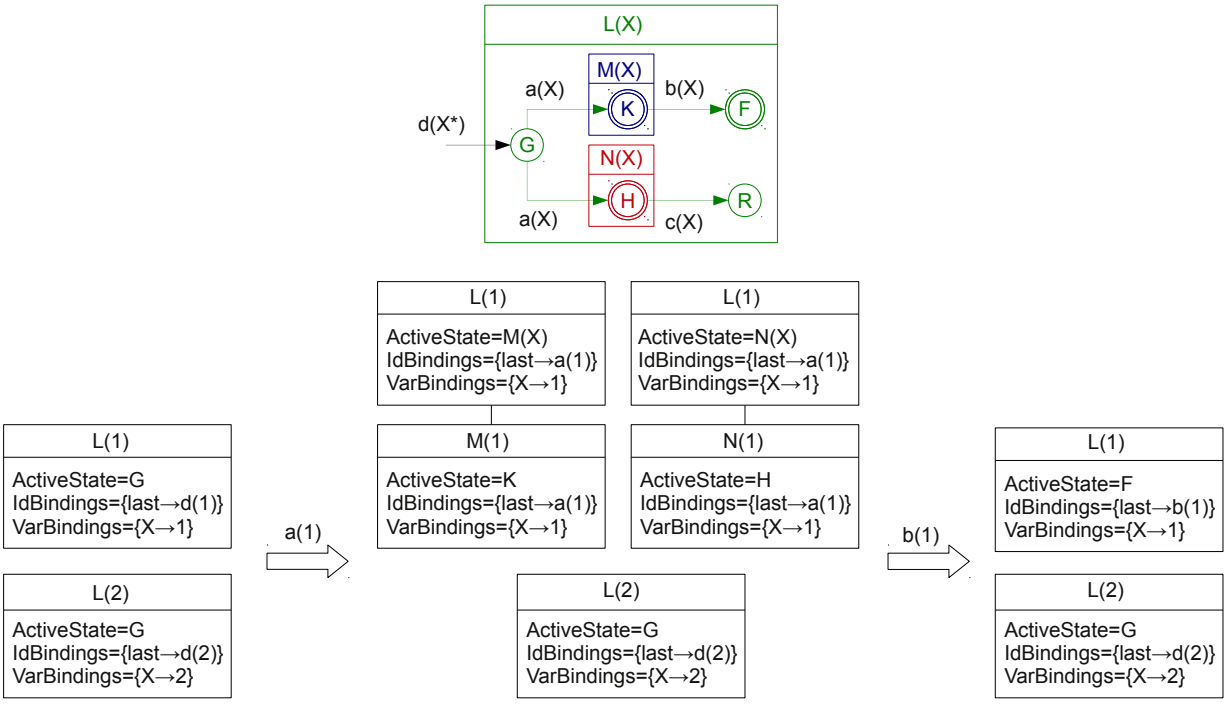

Figure 6.11: Set of nondeterministic runs

empty, i.e. no run was able to accept an event, the stream violates the IHTA. (Listing 6.1 is devoted to the stream verification algorithm which is based on this idea.) The reason of run branching (instead of state merging) is that each of nondeterministic runs can create new instances.

Example: Consider Figure 6.11. Initially there are two runs of the IHTA. Assume $a(1)$ arrives. It is relevant for the first run. The run is branched into two nondeterministic runs. The (same!) instance $L(1)$ has different active states and different subinstances in different runs. If at least one of the nondeterministic runs is successful, the stream does not violate the IHTA.

The second difference between classical nondeterminism [65] and nondeterminism of IHTA is that the elements of an input word (i.e. events of a stream) are not necessarily ordered, i.e. they can have the same occurrence time since this is an important feature in many practical applications. The number of events with the same occurrence time must be finite, otherwise there would be no time progress. This is no restriction of practical applications because the number of events with the same occurrence time is unbounded. The order in which such events are matched by transitions of an instance is ambiguous, and a run evolves differently depending on a particular order.

IHTA handle this by keeping track of all possible choices of event order. For a set of n events with equal occurrence time there are $n!$ (the factorial of n) possible permu-

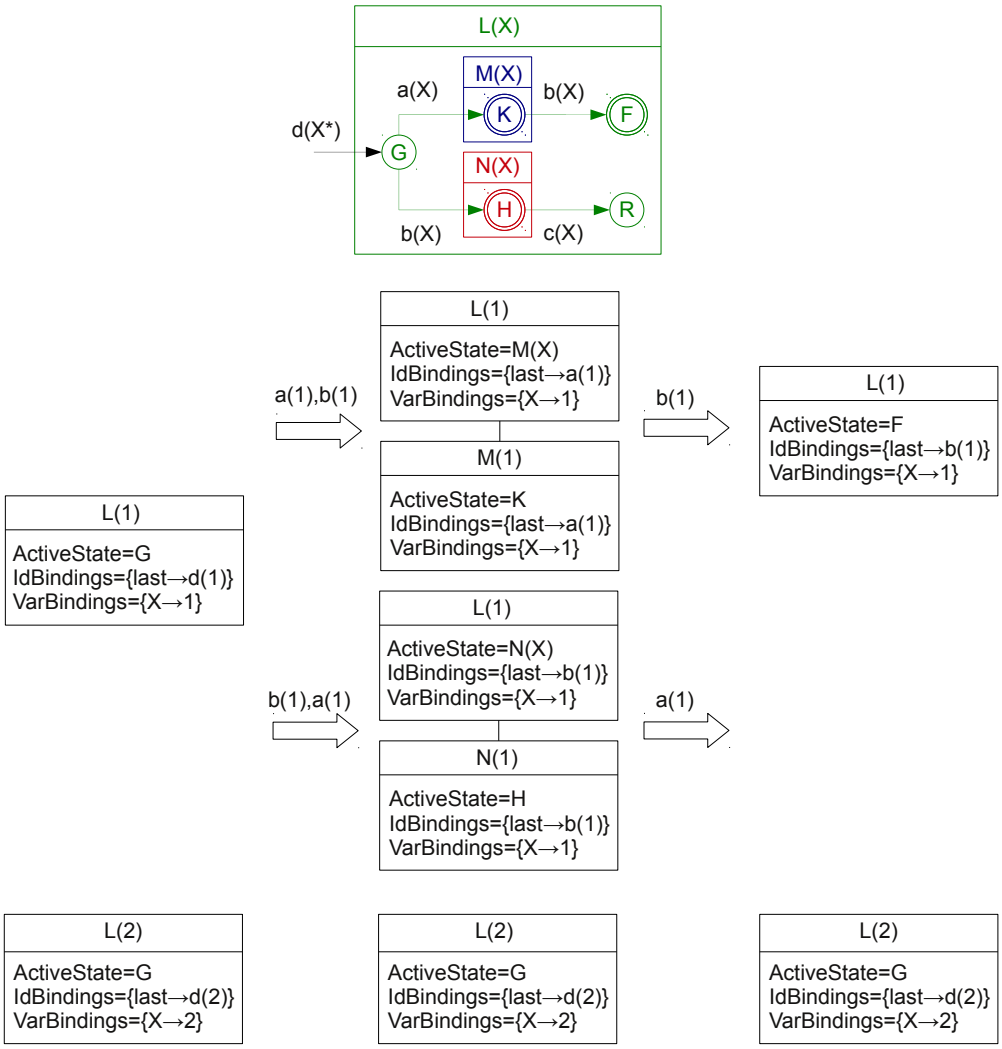

Figure 6.12: Set of nondeterministic runs

tations. Therefore, when such a set is detected, every existing run for which the set is relevant is branched into $n!$ runs. Each of these runs separately handles one possible event permutation and is dropped when it cannot match an event.

Example: Consider Figure 6.12. Initially there are two runs of the IHTA. Assume two events $a(1)$ and $b(1)$ arrive at the same time. They are relevant for the first run. The run is split into two nondeterministic runs. The (same!) instance $L(1)$ has different active states and different subinstances in different runs. The first one of the new runs works on $a(1), b(1)$, the second on $b(1), a(1)$. The second run is not able to accept $a(1)$ and is deleted. The first run accepts both events.

All possible permutations of events with the same end time point of their occurrence times must be considered if these events are relevant for the same run (not only for the same instance!). The order of execution of events which are relevant for different runs

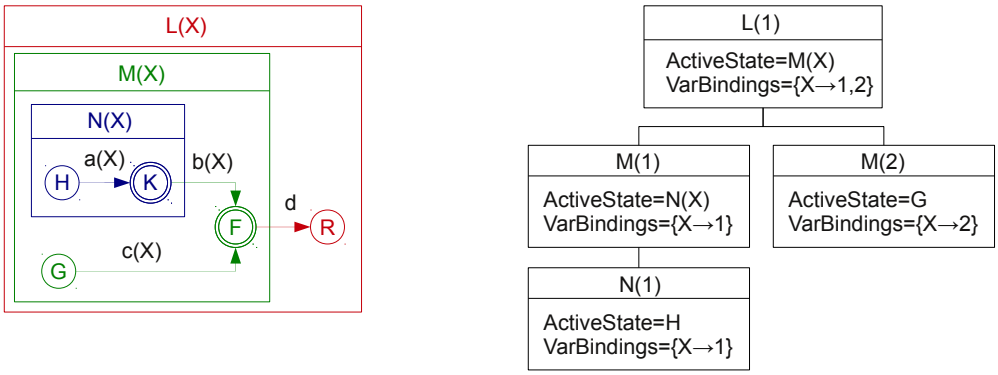

Figure 6.13: The (incomplete) IHTA and its run.

does not matter.

Example: Consider Figure 6.13. Assume events $a(1), b(1), c(2), d$ arrive at the same time. They are relevant for the same run but for four different instances in the run. These events must be processed in the above or alternatively in the following orders: $a(1), c(2), b(1), d$ or $c(2), a(1), b(1), d$. The evaluations of these permutations return the same run. A set of nondeterministic runs (Definition 17) saves only different runs. All other permutations of the events cannot be accepted. If only one permutation p of the events were considered and if p were not the one of the above permutations, the stream verification algorithm (Listing Listing 6.1) would stop the evaluation by returning false.

Definition 17 (**Set of Nondeterministic Runs**)**.** Let r be a run of IHTA. Let I_r be the set of instances in r. Let $t_1, \ldots, t_k \in T$ be the set of all transitions enabled in any instance $i \in I_r$. Then r is replaced by the set of k non-deterministic runs $r_1, \ldots, r_k$ (which belong to the same set of nondeterministic runs as r) such that $r_n = \mathcal{D}_{i_n,t_n}(r)$ where $r_n \in \{r_1, \ldots, r_k\}$ and $t_n \in \{t_1, \ldots, t_k\}$ is a transition enabled in $i_n \in I_r$.

Definition 18 (**Automaton Configuration**)**.** Let $\mathcal{I}$ be IHTA. Automaton configuration of $\mathcal{I}$ is the set of all sets of nondeterministic runs of $\mathcal{I}$.

Definition 19 (**Event Accepted by an Instance**)**.** Let E be an event stream and $e \in E$. Let $\mathcal{I}$ be IHTA and i be an instance of some non-atomic state of $\mathcal{I}$. e is accepted by i if there is a transition $t \in T$ and such that t is enabled by e in i.

Definition 20 (**Event Accepted by a Run**)**.** Let E be an event stream and $e \in E$. Let $\mathcal{I}$ be IHTA and r be a run of $\mathcal{I}$. Let I_r be the set of instances in r. e is accepted by r if e is accepted by all instances $i \in I_r$ for which e is relevant.

Definition 21 (**Event Stream Accepted by IHTA**)**.** Let $E[\leq p]$ be an event stream arrived until a time point $p \in \mathbb{T}$. Let $\mathcal{I}$ be IHTA. If at least one run r of each set of

nondeterministic runs of $\mathcal{I}$ accepts all events of $E[\leq p]$ which are relevant for r and is in an atomic end state of $\mathcal{I}$, $E[\leq p]$ is accepted by $\mathcal{I}$.

Let $\mathcal{A}$ be automaton configuration. At the beginning $\mathcal{A}$ is empty (compare Line 1 in Listing 6.1). $\mathcal{A}$ is modified according to the automaton configuration modification algorithm in Listing 6.1. The pseudo code of the algorithm is followed by its detailed line-by-line explanation.

Listing 6.1: Automaton Configuration Modification Algorithm

```
1  A = ∅
2  forever
3    events ← get_next_events()
4    if events = ∅
5    then for each ndRuns ∈ A
6           new_ndRuns ← ∅
7           for each run ∈ ndRuns
8             new_ndRuns ← new_ndRuns ∪ DelayTransitions(run)
9           end
10          A ←(A − ndRuns) ∪ new_ndRuns
11        end
12   else for each event ∈ events
13          for each enter t ∈ T enabled by event
14            new_run ← Initialize_t()
15            for each i ∈ I_new_run
16              set_accepted(event, i)
17            end
18            new_ndRuns ← new_run
19            A ←A ∪ new_ndRuns
20          end
21        end
22        for each ndRuns ∈ A
23          new_ndRuns ← ∅
24          for each run ∈ ndRuns
25            for each permutation ∈ permutations(events)
26              new_ndRuns ← new_ndRuns ∪ EventTransitions(run, permutation)
27            end
28          end
29          if new_ndRuns = ∅
30          then return false
31          else A ←(A − ndRuns) ∪ new_ndRuns
32          end
33        end
34   end
35 end
36 function DelayTransitions(run)
37   new_runs ← ∅
38   for each i ∈ I_run
39     for each t ∈ T enabled in i
```

```
40          new_run ← 𝒟_{i,t}(run)
41          new_runs ← new_runs ∪ DelayTransitions(new_run)
42        end
43    end
44    if new_runs = ∅
45    then return run
46    else return new_runs
47    end
48 end
49 function EventTransitions(run,events)
50    new_runs ← ∅
51    relevant_events ← events
52    for each event ∈ relevant_events
53      if event is relevant for run
54      then for each i ∈ I_run
55              for each t ∈ T enabled in i
56                if t is enabled by event
57                then if ¬get_accepted(event,i)
58                     then set_accepted(event,i)
59                          new_run ← 𝒟_{i,t}(run)
60                          new_runs ← new_runs ∪ EventTransitions(new_run,events)
61                     end
62                else new_run ← 𝒟_{i,t}(run)
63                     new_runs ← new_runs ∪ EventTransitions(new_run,events)
64                end
65             end
66             if event is relevant for i and ¬get_accepted(event,i)
67             then new_runs ← ∅
68                  return new_runs
69             end
70           end
71      else relevant_events ← relevant_events − event
72      end
73    end
74    if relevant_events = ∅
75    then return run
76    else return new_runs
77    end
78 end
```

The algorithm in Listing 6.1 runs until the event stream violates the IHTA with the automaton configuration $\mathcal{A}$. Otherwise the algorithm does not terminate. An iteration of the main loop in lines 2–35 takes place every time when either there is at least one incoming event, i.e. $events \neq \emptyset$ (line 3), or at least one delay transition can be executed.

If there are no incoming events (line 4), only delay transitions can be processed (lines 5–11). For each set of nondeterministic runs $ndRuns$ and for each run in it all enabled delay transitions are fired by calling the function $DelayTransitions(run)$ in line 8. Since delay transitions can be performed nondeterministically, i.e. when in one state more than one delay transitions are triggered at the same time, the run can be split into several nondeterministic runs (compare Definition 17). All of them are accumulated in the set of new nondeterministic runs new_ndRuns in line 8. Finally, the old set of nondeterministic runs $ndRuns$ of the run is replaced by the new set of nondeterministic runs new_ndRuns in the automaton configuration $\mathcal{A}$ in line 10.

If there are incoming events, both delay and event transitions can be processed (lines 12–34). First in lines 12–21, all enter transitions enabled by the input events are triggered. These transitions create new runs as defined by Definition 14 in line 14. These new runs are added to the automaton configuration $\mathcal{A}$ in line 19. Since an event is processed by an instance only once, line 16 saves that each newly created instance already processed the event which had triggered the respective enter transition. Note that this first step is the only step of the algorithm which is possible when the main loop in lines 2–35 is called for the first time with $\mathcal{A} = \emptyset$. If at the end of this first step, $\mathcal{A}$ contains newly created runs, the second step described in the following becomes possible.

Second in lines 22–33, the algorithm checks for each run of each set of nondeterministic runs whether the run accepts each relevant event (Definition 20) of at least one permutation of the incoming events by calling the function $EventDelayTransitions(run, permutation)$ for each run and each $permutation$ of events in line 26. If it is the case, this run is replaced by the resulting new runs, otherwise this run is deleted. This is explained in more detail in the following.

Let $events$ be the set of incoming events with the same end time point of their occurrence times. Assume all $events$ are relevant for the same run. As motivated above all possible permutations of $events$ must be considered. There are $|events|!$ (the factorial of $|events|$) permutations where $|events|$ denotes the number of $events$.

For each set of nondeterministic runs $ndRuns$ (line 22), for each run in it (line 24), and for each $permutation$ of the incoming events (line 25), it is tested whether the run accepts all relevant events of the $permutation$ (all respective enabled event or delay transitions are fired) in line 26. The set new_ndRuns accumulates all new runs resulting from the

runs which were able to accept all relevant events in at least one permutation. If no *run* is able to accept all the incoming events in any *permutation*, the set *new_ndRuns* is empty. Otherwise the set *new_ndRuns* contains new nondeterministic runs.

If the set *new_ndRuns* is not empty, the set of old nondeterministic runs *ndRuns* of the *run* is replaced by the set of new nondeterministic runs *new_ndRuns* in the automaton configuration $\mathcal{A}$ in line 31. Otherwise the input event stream violates the IHTA with the automaton configuration $\mathcal{A}$. Compare Definition 21. Therefore, the execution of the algorithm is stopped by returning *false* in line 30.

The main loop is based on two auxiliary functions $DelayTransitions(run)$ processing all enabled delay transitions in the *run* and $EventTransitions(run, events)$ processing all enabled event and delay transitions in the *run*. Separate treatment of event transitions is necessary because $EventTransitions(run, events)$ returns a (nonempty) set of new runs only if the *run* accepted all relevant *events*. Otherwise the returned set is empty. $DelayTransitions(run)$, in contrast, always returns a nonempty set of new runs. If no delay transition can fire in the *run*, the unchanged *run* is returned. Both functions are described in more detail in the following.

$DelayTransitions(run)$ takes a single *run* as its argument and returns the set *new_runs* after recursively performing all delay transitions which are possible at the current time point in the *run*.

More exactly: For each instance i in the set of instances I_{run} of the *run* (line 38) and for each delay transition t enabled in i (line 39), t is fired in i in line 40. The result is a *new_run*. Compare Definition 16. Other enabled delay transitions are recursively performed on the *new_run* in line 41. All these nondeterministic runs are accumulated by the set *new_runs* in line 41. If the set *new_runs* is not empty, it is returned in line 46. Otherwise the unchanged *run* is returned in line 45.

$EventTransitions(run, events)$ is similar to $DelayTransitions(run)$ but it additionally takes an ordered list of events, *events*, as an argument and processes all enabled event and delay transitions in the *run*. The function returns a nonempty set of new runs only if all relevant *events* are accepted by the *run*, otherwise the resulting set is empty.

More exactly: For each *event* of the input *events* (line 52) it is tested whether it is relevant for the *run* in line 53. If it is the case then for each instance i in the set of instances I_{run} of the *run* (line 54) and for each event or delay transition t enabled in i (line 55), t is fired in i in line 59 or line 62 depending on whether t is an event or a delay transition (see below). The result is a *new_run*.

If t is a delay transition, it is treated as described above (compare lines 40–41 with lines

62–63) with the only difference that the new_run must still process all $events$. Therefore, $EventTransitions(new\_run, events)$ is recursively called in line 63.

If t is an event transition (line 56) and the $event$ has not been accepted by the instance i yet (line 57), the $event$ is marked as accepted by i in line 58, t is triggered by the $event$, and the resulting new run is saved in new_run in line 59. The function is called recursively on the new_run in line 60.

If the $event$ is relevant for the instance i in the run but has not been accepted by i (line 66), then the run was not able to accept all events (compare Definition 20) and the resulting set new_runs is empty (line 67–68).

If some $event$ of the input $events$ is not relevant for the run then it is deleted from the list $relevant\_events$ in line 71.

If no event of the input $events$ is relevant for the run (line 74), the set $relevant\_events$ is empty and the unchanged run is returned in line 75. Otherwise in line 76, the set new_runs is returned.

Termination

Let $\mathcal{I}$ be IHTA.

The algorithm in Listing 6.1 does intentionally not terminate. However, each iteration of the main loop in lines 2–35 terminates under the following conditions:

1. For each time point t, the number of incoming events $|events|$ is finite.
2. For each time point, there must be no cycles of delay transitions in $\mathcal{I}$, i.e. the same delay transition may not be processed more than once by runs yielded from calls of $DelayTransitions(run)$.

The number of transitions $|T|$ is finite. The maximum number of instances j created by an instantiating transition is finite.

The number of sets of nondeterministic runs n_s in the automaton configuration of $\mathcal{I}$ is finite. The total number of runs n_r in all sets of nondeterministic runs is also finite. The total number of instances n_i in all runs is finite. $n_s \in \mathcal{O}(n_i)$ and $n_r \in \mathcal{O}(n_i)$ since in the worst each instance belongs to its own run which is the only run of a set of nondeterministic runs.

Let $|E[\leq p]|$ denote the number of events relevant for $\mathcal{I}$ and arrived on the stream E until the time point $p \in \mathbb{T}$. Then $n_i \in \mathcal{O}(|E[\leq p]| \cdot |E[\leq p]|! \cdot |T| \cdot j)$ because in the worst case each event arrived on the stream E until the time point $p \in \mathbb{T}$ in each permutation of events triggers an instantiating transition $t \in T$ creating at most j instances.

For these reasons the number of iterations of all nested loops of the algorithm is finite.

Complexity

$$
\begin{aligned}
&DelayTransitions(run) && \in \mathcal{O}(n_i \cdot |T|) \\
&EventTransitions(run, events) && \in \mathcal{O}(n_i \cdot |T| \cdot |events|) \\
&main(\mathcal{A}, events) && \in \mathcal{O}(n_i \cdot |events|! \cdot n_i \cdot |T| \cdot |events|) \\
& && \in \mathcal{O}((|E[\leq p]| \cdot |E[\leq p]|! \cdot |T| \cdot j)^2 \cdot |events|! \cdot |T| \cdot |events|) \\
& && \in \mathcal{O}(|E[\leq p]|^2 \cdot |E[\leq p]|!^2 \cdot |T|^3 \cdot j^2 \cdot |events|! \cdot |events|)
\end{aligned}
$$

Chapter 7

Conclusion and Future Work

The formalism for expressing application semantics in event-driven applications, called Instantiating Hierarchical Timed Automata (IHTA), is motivated, illustrated by the auction use case, and formally defined in this work. IHTA possess the following core features:

1. IHTA are stateful. State changes are triggered by events and/or delays.
2. IHTA are independent from the language specifying transition labels. Event queries should allow access to event data. Temporal constraints should be expressed on the occurrence time of matched events and/or on the current time. Other kinds of constraints are not considered in this work to keep IHTA readable.
3. IHTA are non-deterministic and able to work with events with the same occurrence time.
4. IHTA are modular which implies abstraction, readabilility, (ex-) changeability, and reuse of the modules.
5. IHTA can represent an arbitrary number of concurrent processes by instantiation of module specifications.

IHTA will be used for semantic optimization of event queries in the following way. Cardinality and functional dependencies between events and states which are expressed by IHTA will be represented as a set of constraints in the Event Stream Constraint Language ESCL [62]. This transformation must be proven to be correct with respect to the formal semantics of IHTA defined in this work. The reasons for this transformation are the following: First, the method for the semantic optimization of database queries which is called the residue method [21] and based on constraints can be adapted to the peculiarities of CEP such as the validity time of constraints and the evaluation time of

queries. The existing research results will be reused in this way to achieve new goals. Besides, it is possible to provide additional user-defined constraints not expressed by IHTA, for example constraints on the data carried by events or states.

In [4], Timed Finite Automata have been studied from the perspective of formal language theory, considering closure properties, decision problems and subclasses for both deterministic and nondeterministic transitions with both Büchi and Muller acceptance conditions. No attempt has been made to study IHTA in a similar way until now. It is presumed, however, that the theoretical insights gained from Timed Automata are not fully applicable to IHTA due to the following reasons: First, the state space of IHTA is unbounded since the number of concurrent instances is unbounded. Second, the temporal constraints of IHTA (expressed on event occurrence time) are more powerful than that of Timed Automata (expressed on local clocks) because only access to the end of event occurrence time can be simulated by local clocks but not access to the *beginning* of event occurrence time. Studying properties of IHTA from a formal language theory perspective is a possible subject for future work.

Bibliography

[1] Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler. Scheduling with Timed Automata. *Theor. Comput. Sci.*, 354:272–300, March 2006.

[2] Parosh Aziz Abdulla and Aletta Nylén. Timed Petri Nets and BQOs. In *Applications and Theory of Petri Nets*, volume 2075 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2001.

[3] Rajeev Alur. Timed Automata. In *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer, 1999.

[4] Rajeev Alur and David L. Dill. Automata For Modeling Real-Time Systems. In *Proc. of the* 17^{th} *Int. Colloquium on Automata, Languages and Programming*, ICALP '90, pages 322–335. Springer, 1990.

[5] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126:183–235, April 1994.

[6] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating Hierarchical State Machines. In *Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 169–178. Springer, 1999.

[7] Rajeev Alur and P. Madhusudan. Decision Problems for Timed Automata: A Survey. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2004.

[8] Rajeev Alur and Mihalis Yannakakis. Model Checking of Hierarchical State Machines. *SIGSOFT Softw. Eng. Notes*, 23:175–188, November 1998.

[9] Rajeev Alur and Mihalis Yannakakis. Model Checking of Message Sequence Charts. In *CONCUR'99 Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 1999.

[10] Danièle Beauquier. Muller Automata and Bi-infinite Words. In *Fundamentals of Computation Theory*, FCT '85, pages 36–43. Springer, 1985.

[11] Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2004.

[12] Marilyn Bohl and Maria Rynn. *Tools for Structured and Object-Oriented Design.* Prentice Hall, 2008.

[13] Benedikt Bollig and Dietrich Kuske. Distributed Muller Automata and Logics. *Information and Computation*, 206, 2008.

[14] Anthony J. Bonner and Michael Kifer. Transaction Logic Programming (or, A Logic of Procedural and Declarative Knowledge), 1995.

[15] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-Oriented Analysis and Design with Applications,* 3^{rd} *ed.* Addison-Wesley Professional, 2007.

[16] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide,* 2^{nd} *ed.* Addison-Wesley Professional, 2005.

[17] Patricia Bouyer and François Laroussinie. *Model Checking Timed Automata*, pages 111–140. ISTE, 2010.

[18] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A Model-checking Tool for Real-time Systems. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998.

[19] François Bry and Michael Eckert. Rule-Based Composite Event Queries: The Language $XChange^{EQ}$ and its Semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*, volume 4524 of *Lecture Notes in Computer Science*, 2007.

[20] Julius Richard Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Proc. Int. Congress on Logic, Method, and Philosophy of Science*, pages 1–12. Standford University Press, 1960.

[21] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based Approach to Semantic Query Optimization. volume 15, pages 162–207. ACM, 1990.

[22] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28:114–133, January 1981.

[23] Ashok K. Chandra and Larry J. Stockmeyer. Alternation. In *Proc.* 17^{th} *IEEE Symp. on Foundations of Computer Science*, pages 98–108, 1976.

[24] Kwang Ting Cheng and A. S. Krishnakumar. Automatic Functional Test Generation Using the Extended Finite State Machine Model. In *Proc. of the* 30^{th} *Int. Design Automation Conf.*, DAC '93, pages 86–91. ACM, 1993.

[25] O. Coudert, C. Berthet, and J.C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, 1989.

[26] Alexandre David. *Hierarchical Modeling and Analysis of Timed Systems*. PhD thesis, Uppsala University, November 2003.

[27] Alexandre David and M. Oliver Möller. From HUPPAAL to UPPAAL: A Translation from Hierarchical Timed Automata to Flat Timed Automata. Technical Report RS-01-11, BRICS, March 2001.

[28] Alexandre David and Yi Wang. Hierarchical Timed Automata for UPPAAL. In 10^{th} *Nirdic Workshop on Programming Theory*. Turku Centre for Computer Science, 1998.

[29] René David and Hassane Alla. Petri Nets for Modeling of Dynamic Systems: A Survey. *Automatica*, 30:175–202, February 1994.

[30] Martin Dickhöfer and Thomas Wilke. Timed Alternating Tree Automata: The Automata-Theoretic Solution to the TCTL Model Checking Problem. In *Proc. of the* 26^{th} *Int. Colloquium on Automata, Languages and Programming*, pages 281–290. Springer, 1999.

[31] Cătălin Dima and Ruggero Lanotte. Distributed Time-Asynchronous Automata. In *Theoretical Aspects of Computing*, volume 4711 of *Lecture Notes in Computer Science*, pages 185–200. Springer, 2007.

[32] Luping Ding, Songting Chen, Elke A. Rundensteiner, Junichi Tatemura, Wang-Pin Hsiung, and K. Selcuk Candan. Runtime Semantic Query Optimization for Event Stream Processing. In *Proc. Int. Conf. on Data Engineering*, pages 676–685. IEEE Computer Society, 2008.

[33] Michael Eckert. *Complex Event Processing with XChange^{EQ}: Language Design, Formal Semantics and Incremental Evaluation for Querying Events*. Dissertation/Ph.D. thesis, Institute for Informatics, University of Munich, 2008.

[34] Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. *Two Semantics for CEP, no Double Talk: Complex Event Relational Algebra (CERA) and its Application to XChange^{EQ}*. Springer, 2010. to appear.

[35] Reiner Fehling. A Concept of Hierarchical Petri Nets with Building Blocks. In *Advances in Petri Nets*, volume 674 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 1993.

[36] Olivier Finkel. Undecidable Problems about Timed Automata. In *Formal Modeling and Analysis of Timed Systems*, volume 4202 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2006.

[37] N.A. Fountas, N.D. Hatziargyriou, and Valavanis K.P. Hierarchical Time-extended Petri Nets as a Generic Tool for Power System Restoration. volume 12, pages 837–843. IEEE Computer Society, 1997.

[38] Mark A. Fryman. *Quality and Process Improvement*. Cengage Learning, 2002.

[39] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata. In *Proc. Int. Conf. on Database Theory*, pages 173–189. Springer, 2002.

[40] Olga Grinchtein. *Learning of Timed Systems*. Dissertation/Ph.D. thesis, Uppsala University, 2008.

[41] Olga Grinchtein, Bengt Jonsson, and Martin Leucher. Learning of Event-recording Automata. In *Theoretical Computer Science*, volume 411, pages 4029–4054. Elsevier Science Publishers Ltd., 2010.

[42] Alexander Grosskopf, Gero Decker, and Mathias Weske. *The Process: Business Process Modeling using BPMN*. Meghan Kiffer Press, 2009.

[43] Hermann Gruber, Markus Holzer, Astrid Kiehn, and Barbara König. On Timed Automata with Discrete Time – Structural and Language Theoretical Characterization. In *Developments in Language Theory*, volume 3572 of *Lecture Notes in Computer Science*, pages 23–47. Springer, 2005.

[44] David Harel. Statecharts: A Visual Formalism for Complex Systems. volume 8. Elsevier Science Publishers Ltd., 1987.

[45] David Hemer, Robert Colvin, Ian J. Hayes, and Paul A. Strooper. Don't Care Nondeterminism in Logic Program Refinement. *Electronic Notes in Theoretical Computer Science*, pages 101–121, 2002.

[46] Martijn Hendriks. *Model Checking Timed Automata: Techniques and Applications.* Dissertation/Ph.D. thesis, Radboud University Nijmegen, 2006.

[47] R. Hojatic, V. Singhal, and Robert K. Brayton. Edge-streett/edge-rabin automata environment for formal verification using language containment. Technical Report UCB/ERL M94/12, EECS Department, University of California, Berkeley, 1994.

[48] Farrell Joyce. *Programming Logic and Design,* 5^{th} *ed. Comprehensive.* Thomson, 2008.

[49] Ekkart Kindler. On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. Research report, Computer Science Department, University of Paderborn, Germany, 2006.

[50] Valerie King, Orna Kupferman, and Moshe Y. Vardi. On the Complexity of Parity Word Automata. In *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 276–286. Springer, 2001.

[51] Sebastian Kupferschmid. *Directed Model Checking for Timed Automata.* Dissertation/Ph.D. thesis, Albert-Ludwigs University Freiburg, 2010.

[52] F. Laroussinie, N. Markey, and Ph. Schnoebelen. Model Checking Timed Automata with One or Two Clocks. In *Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 387–441. Springer, 2004.

[53] Slawomir Lasota and Igor Walukiewicz. Alternating Timed Automata. *ACM Trans. Comput. Logic*, 9:1–27, April 2008.

[54] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A Transducer-Based XML Query Processor. In *Proc. Int. Conf. on Very Large Data Bases*, pages 227–238. VLDB Endowment, 2002.

[55] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Trans. Softw. Eng. Methodol.*, 11:2–57, January 2002.

[56] H. Oswald, R. Esser, and R. Mattmann. An environment for specifying an executing hierarchical petri nets. In *Proc. of the* 12^{th} *Int. Conf. on Software Engineering*, pages 164–172. IEEE Computer Society Press, 1990.

[57] Christos Papadimitriou. *Alternation*, Section 16.2, pages 399–401. Addison Wesley, 1993.

[58] Dominique Perrin and Jean-Eric Pin. *Infinite Words: Automata, Semigroups, Logic and Games*. Elsevier, 2004.

[59] James L. Peterson. Petri Nets. *ACM Comput. Surv.*, 9:223–252, September 1977.

[60] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*. Dissertation/Ph.D. thesis, Department of Computer Systems, Uppsala University, 1999.

[61] Nir Piterman. From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata. IEEE Computer Society, 2006.

[62] Olga Poppe and François Bry. The Grammar and the Geclarative Semantics of ESCL. Research report PMS-FB-2011-1, Institute for Informatics, University of Munich, 2011.

[63] S. Ramaswamy and K.P. Valavanis. Hierarchical Time-extended Petri Nets (H-EPNs) Based Error Identification and Recovery for Multilevel Systems. volume 26, pages 164–175. IEEE Computer Society, 1996.

[64] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed. On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. In *Proc. of the* 3^{rd} *Asia-Pacific Conf. on Conceptual Modelling*, volume 53 of *APCCM*, pages 95–104. Australian Computer Society, Inc., 2006.

[65] Uwe Schöning. *Theoretische Informatik - kurzgefasst*. Spektrum Akademischer Verlag, 2003.

[66] Michael Sipser. *Alternation*, Section 10.3, pages 380–386. PWS Publishing, 2006.

[67] Maria Sorea. Bounded Model Checking for Timed Automata. In *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier Science Publishers Ltd., 2002.

[68] Hong Su, Elke A. Rundensteiner, and Murali Mani. Semantic Query Optimization in an Automata-Algebra Combined XQuery Engine over XML Streams. In *Proc. Int. Conf. on Very Large Data Bases*, pages 1293–1296. VLDB Endowment, 2004.

[69] Hong Su, Elke A. Rundensteiner, and Murali Mani. Semantic Query Optimization for XQuery over XML Streams. In *Proc. Int. Conf. on Very Large Data Bases*, pages 277–288. VLDB Endowment, 2005.

[70] Wolfgang Thomas. Automata on Infinite Objects. In *Handbook of Theoretical Computer Science*, pages 133–164. Elsevier, 1990.

[71] Salvatore La Torre and Margherita Napoli. Timed Tree Automata with an Application to Temporal Logic. *Acta Informatica*, 38:89–116, 2001.

[72] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[73] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. An Algorithm for Learning Real-time Automata. In *Proc. of the* 18^{th} *Benelearn*, 2007.

[74] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. One-Clock Deterministic Timed Automata Are Efficiently Identifiable in the Limit. In *Language and Automata Theory and Applications*, volume 5457 of *Lecture Notes in Computer Science*, pages 740–751. Springer, 2009.

[75] Sicco Ewout Verwer. *Efficient Identification of Timed Automata: Theory and Practice*. Dissertation/Ph.D. thesis, Delft University of Technology, 2010.

[76] Jiacun Wang. *Timed Petri Nets: Theory and Application*. Springer, 1998.

[77] W.M.Zuberek. Timed Petri Nets: Definitions, Properties, and Applications. *Microelectronics Reliability*, 31:627–644, 1991.

[78] Mihalis Yannakakis. Hierarchical State Machines. In *Theoretical Computer Science: Exploiting new frontiers of theoretical informatics*, volume 1872 of *Lecture Notes in Computer Science*, pages 315–330. Springer, 2000.

[79] Sergio Yovine. Model Checking Timed Automata. In *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 114–152. Springer, 1998.

[80] Xudong Zhao and Yulin Feng. Automatic and Hierarchical Verification for Concurrent Systems. volume 5, pages 241–249. Springer, 1988.

Index