Event Actors Based Approach for Supporting Analysis and Verification of Event-Driven Architectures

Huy Tran and Uwe Zdun
Software Architecture Research Group
University of Vienna, Austria.
Email: huy.tran|uwe.zdun@univie.ac.at

***Abstract*—Event-based communication styles are potential solutions for facilitating high flexibility, scalability, and concurrency of distributed systems due to the intrinsic loose coupling of the participants. However, software developers often find the event-driven communication style unintuitive, especially for large and complex systems with numerous constituting elements, because of its non-deterministic characteristics. In this paper, we propose a novel approach based on DERA—an event actor-based framework—which can be used to describe distributed event-based systems with reduced nondeterminism. DERA's graphical notations support representing a current snapshot of an event-based system closely to the intuitive perception of the developers. We propose a formal specification of the event actors-based constructs and the graphical notations based on Petri nets in order to enable formal analysis of such snapshots. Based on this, an automated translation from event actors-based constructs to Petri nets using template-based model transformation techniques is also developed. The applicability of our approach is shown through an industrial case study in the field of service platform integration.**

***Keywords*-event-driven architecture; event actors; DERA; verification; Petri nets;**

I. Introduction

Event-driven architectures are promising for modeling and developing loosely coupled systems to facilitate high flexibility, scalability, and concurrency [1]. Due to the inherent loose coupling of the participants and the clear separation of communication from computation, the event-based architectural style has become a viable solution for supporting the integration of heterogeneous components to form complex distributed software systems. The advantages of event-driven communication styles have been extensively recognized and exploited both in academia and industry, resulting in a substantial amount of work in different domains such as middleware infrastructure [2], event-based coordination [3], active database systems [4], and service-oriented architectures [5], to name but a few.

Unfortunately, apart from introducing additional degrees of freedom and more flexibility, the loose coupling in event-based systems also increases the complexity of developing and understanding these systems. Due to its non-deterministic characteristics, software developers often find the event-driven communication style unintuitive, especially for large and complex systems with plenty of elements.

In order to analyze an event-based software system, the developers often need to obtain an adequate representation of the system. Unfortunately, achieving an adequate representation of an event-based system, which is close to the perception of the developers, for instance, as that of UML diagrams, is not well-supported. Normally, the developers have to investigate the source code to determine the sources and targets of the events exchanged among highly decoupled components of the system. Next, the representation has to be translated to some formal specifications that can be used by verification tools. To the best of our knowledge, there are no previous attempts in literature on supporting the developers in performing the aforementioned activities.

Our previous work [6] presented the DERA (dynamic event-driven actors) approach that aims at enabling runtime flexibility for supporting runtime evolution and dynamic adaptation while reducing the non-deterministic nature of event-based systems. In particular, DERA encapsulates execution elements (e.g., activities, tasks, functions) in event actors that pose explicit, formally specified interfaces. The event-based communication style is exploited to loosen the dependencies among actors. The interfaces of actors are formally specified and constrained to enable the support for altering actors at runtime (e.g., replacing actors or changing their execution order). In addition, well-defined interfaces of actors also target at reducing the non-deterministic behavior of event-based communication.

In this paper, we propose a novel approach to supporting the analysis and verification of event-driven architectures based on DERA. Firstly, a current snapshot of an event-driven system described using DERA concepts is taken and visualized based on the well-defined interfaces of the event actors and the graphical notations provided in DERA. Then, DERA constructs are described using Petri nets—which is a well-known graphical and mathematical modeling tool for distributed systems [7].

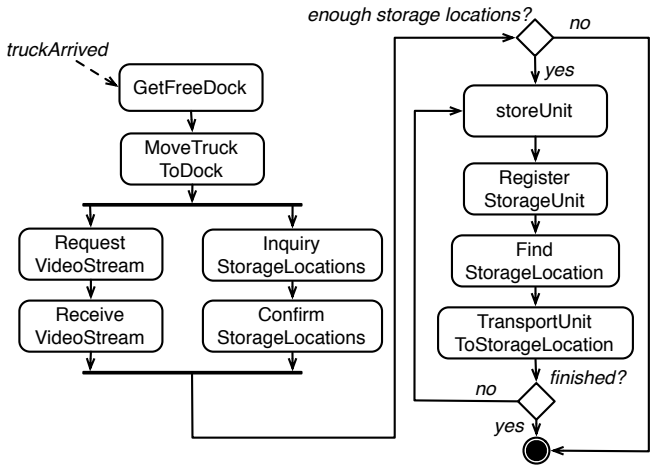


Figure 1. Schematic view of the WH operator's behavior

As a result, we can leverage a multitude of powerful techniques and tools that have been developed for Petri nets for performing formal analysis of a snapshot of the underlying event-driven software system at a certain point in time. We will show the applicability of our approach through an industrial case study in the field of service platform integration.

The paper is structured as following. A running example extracted from the industrial case study is described in Section II. Then we outline the dynamic event-driven actors (DERA) approach in Section III and present our approach to capturing a snapshot of an event-based system described using DERA. Section IV explains in detail our Petri nets-based approach that we leverage for formalizing and analyzing the snapshots of a DERA system. In Section V, we discuss the related work. Section VI closes the paper with contribution summary and future directions.

II. Running Example

We illustrate the application of our approach in this paper using a running example based on an operator application for a warehouse (WH) extracted from an industrial case study in the field of service platform integration[1]. In the context of this application, there are three different domain-specific service platforms, namely, a yard management system (YMS), a warehouse management system (WMS), and a remote monitoring system (RMS) that provide a wide variety of services. The operator application shall utilize these services to perform various necessary tasks that are triggered by events such as the arrival of a truck that carries the products to be stored in the warehouse.

[1]See http://indenica.eu/fileadmin/INDENICA/user_upload/d51-casestud.pdf

When a truck arrives in the yard, the YMS sends a message to triggers the execution of the WH operator application. While the process is running, the operators need live video streams from the cameras provided by the RMS in order to monitor every action happening in the warehouse. Before starting to store products in the warehouse, the operators must inquire if there are enough storage locations. Afterwards, the operators shall instruct the unloading of products from the truck, registering those products in the storage records, transporting them with the conveyor belts, and storing them in the racks. The corresponding behavior of the operator application in reality is very complex. In this paper, for demonstration purpose, we leverage a simplified excerpt of the WH operator's behavior and present in Figure 1 in terms of a UML activity diagram.

This activity diagram presents a typical composition of tasks for the operator application. However, in this domain often changes are required at design time (e.g., each customer and each warehouse and yard are different and require slightly different configurations) or even runtime (e.g., the operator must handle exceptional situations or failures). It is very difficult to model all possible variations and exceptions in one activity diagram. Here, the loose coupling in event-based systems can help to flexibly react to design time or runtime changes. Our approach aims to offer this flexibility together with the possibility to predefine and visually show a typical flow based on event actors that can flexibly be adapted to new situations. The original and the adapted flows implied by the event actors can be verified for avoiding undesired properties such as deadlocks or livelocks.

III. Event-based System Development using DERA

A. Basic concepts

In Figure 2 we present a meta-model of the primitive concepts forming a DERA system. In summary, the central notions of DERA are *events* and *event actors*. An event can be considered essentially as "any happening of interest that can be observed from within a computer" [1] (or a software system). An event may also have some additional attributes such as its unique identifier, correlation identifiers, and so forth.

In DERA, the abstraction of a computational or data handling unit, for instance, executing a service invocation, or accessing and transforming data, to name but a few, is an *event actor* (or *actor* for short). In general, each actor has a well-defined interface represented by the `ActorInterface` class. The actor's interface defines a set of events that the actor awaits (aka input events) and a set of events that the actor will emit after finishing its execution (aka output events). The function of each actor is defined through a concrete instance of the `Behavior`

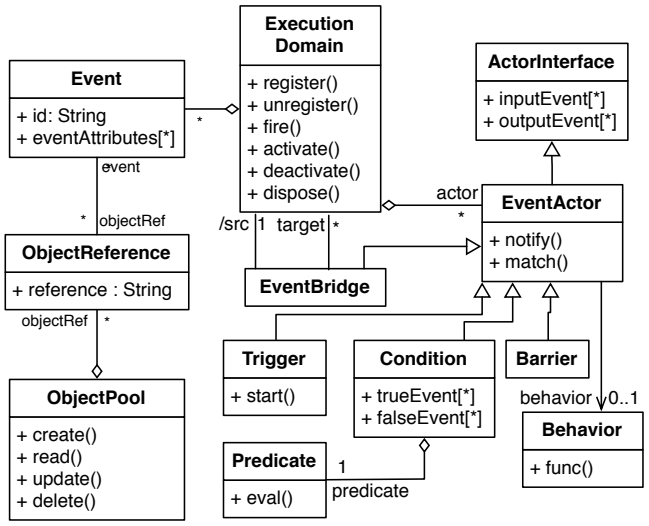


Figure 2. Meta-model for the primitive DERA concepts and their relationships [6]

Table I
DERA GRAPHICAL NOTATIONS

EventActor	Barrier	Condition	Trigger	EventBridge
				

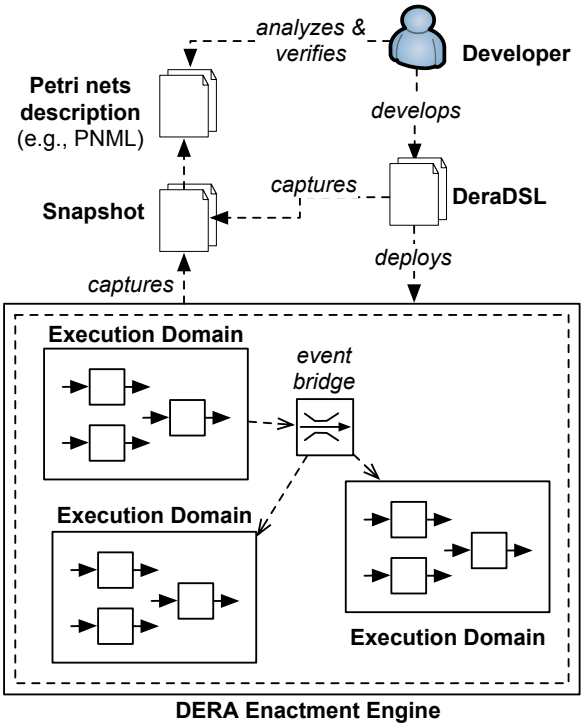


Figure 3. DERA development tool chain and system architecture

class. The execution of an actor's `Behavior`, which is triggered by the arrival of the input events, is not allowed to alter its interface. At the end of its execution, the actor will emit its output events that, in turn, may trigger other actors. DERA also provides primitive constructs, namely, `Condition` and `Barrier` for describing exclusive choices and synchronizations, respectively. A `Trigger` is a special class of event actors that does not need input events. Thus, we can use a `Trigger` to emit events that, in turns, will initiate the executions of other actors. An `EventBridge`, which can forward the incoming events to target execution domains, can be used to connect two domains. The graphical notations of the primitive event actors are shown in Table I. Further details of the aforementioned DERA concepts can be found in [6].

The most significant advantage over traditional event-driven architectures is that DERA enables us to obtain a snapshot of the current state of the system and derive a graph that comprises event actors "*virtually*" connected via their input and output events at design time or runtime. As a result, we are able to support monitoring as well analysis of important properties such as reachability (safety or deadlock checking), liveness, performance, and quality of services of the underlying systems. In the subsequent sections, we will explain how the snapshots of an event-based system described using DERA can be achieved and leveraged for formal analysis and verification using a Petri nets-based approach.

In Figure 3, we present a "big picture" of the DERA development tool chain and runtime architecture. The developer can specify an event actors-based application using the DeraDSL that provides the concepts shown in Figure 2. The DeraDSL code will be then deployed and enacted in a DERA engine. A snapshot of a DERA-based application can be obtained at design time by analyzing the DeraDSL code or at runtime by querying the DERA engine. The snapshot will then be automatically translated to Petri nets descriptions, for instance, the Petri net Markup Language (PNML) used in this paper. The developer can leverage existing tools to analyze and verify the resulting Petri nets description.

B. DERA snapshots

As mentioned above, DERA enables us to capture a snapshot of an event-based system described in terms of event actors that are "*virtually*" connected via events. Such snapshots are the basis for visualizing and analyzing the underlying event-based system.

Definition 1 (DERA snapshot): At a certain point in time, a snapshot $\mathcal{S}$ of a DERA application can be described by a 4-tuple $(A, E, E_{start}, E_{finish})$, where

- A is a finite set of event actors that constitute the functionality of the application. Each event actor can be one of the following types of event actors or

```
module eu.indenica.casestudy.warehouse.operator
domain WarehouseOperator {
 EventActor getFreeDock input[eTruckArrived] output[eGotFreeDock]
 EventActor moveTruckToDock input[eGotFreeDock] output[
  eTruckMoved]
 EventActor requestVideoStream input[eTruckMoved] output[
  eVideoRequested]
 EventActor receiveVideoStream input[eVideoRequested] output[
  eVideoReceived]
 EventActor inquiryStorageLocations input[eTruckMoved] output[
  eInquired]
 EventActor confirmStorageLocations input[eInquired] output[
  eConfirmed]
 Barrier gate input[eConfirmed,eVideoReceived] output[eSynched]
 Condition c1 input[eSynched] when-true[eEnoughLocations]
  when-false[eNotEnoughLocations]
 EventActor storeUnit input[eEnoughLocations, eNotFinished]
  output[eStoreRequested]
 EventActor registerStorageUnit input[eStoreRequested] output[
  eRegistered]
 EventActor findStorageLocation input[eRegistered] output[
  eLocationFound]
 EventActor transportUnitToStorageLocation input[eLocationFound]
  output[eUnitTransported]
 Condition c2 input[eUnitTransported] when-true[eNotFinished]
  when-false[eNotEnoughLocations, eFinished]
 Application WarehouseOperator start-with [eTruckArrived]
  end-with [eFinished]
}
```

Listing 1. DERA description of the warehouse operator

their subtypes: `EventActor`, `Barrier`, `Condition`, `EventBridge`, and `Trigger` (c.f. Figure 2).

- E is a finite set of events published and consumed by the event actors of A;
- $E_{start} \subseteq E$ and $E_{finish} \subseteq E$ are finite sets of events that indicate the start or end of the application respectively.

Based on Definition 1, a snapshot of an event-based system described using DERA can be represented as a graph comprising actors that consequently trigger each other through events. Moreover, we can visualize the snapshot using DERA graphical notations (see Table I). The semantics of the building blocks of a DERA system (i.e., `EventActor`, `Barrier`, and `Condition`) is equivalent to that of the conventional control structures of programming languages whilst their graphical notations are analogous to those of the existing graphical modeling languages that are widely used in both academia and industry for business applications such as BPMN[2] and UML Activity Diagram[3] [6]. Therefore, the visualization of the snapshot of a DERA-based system are also close to the perception of software developers who are used to the aforementioned development concepts and notations.

We present in Listing 1 an excerpt of the warehouse operator system description. The behavior of the warehouse operator is described using the programming constructs provided by DeraDSL [6]. Given the specification of the actors and their interfaces, we can build an

[2]http://www.omg.org/spec/BPMN/2.0/PDF
[3]http://www.omg.org/spec/UML/2.2/Superstructure/PDF

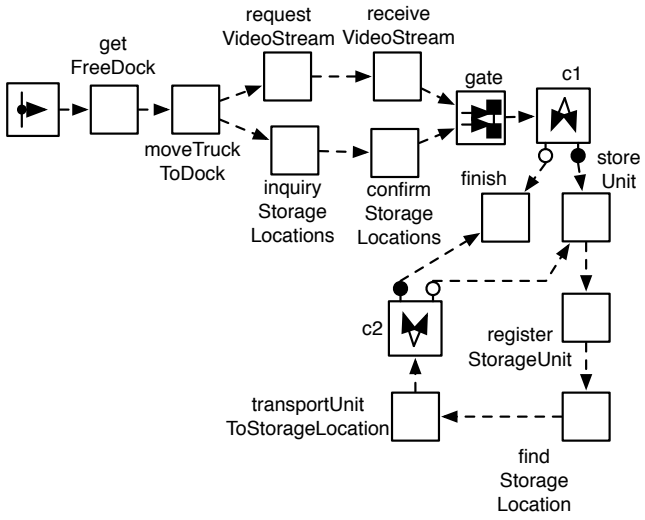


Figure 4. The DERA snapshot of the warehouse operator

intuitive graphical representation of a snapshot of the application (see Figure 4) using the notation defined in [6]. It is important to note that the dashed lines among event actors are not real dependencies but only "virtual" connections achieved by analyzing the inputs and outputs of the actors captured in the snapshot of the application. We can see that the graphical representation shown in Figure 4 is rather similar to the schematic view of the warehouse operator application modeled using a UML activity diagram previously shown in Figure 1. It is our intention to introduce the developers with the concepts and notations that are close to their perceptions that are based on traditional modeling languages instead of proposing new languages and representations that require steep learning-curves.

IV. Verification of DERA-based systems

A. Petri nets introduction

The correctness (i.e., the absence of anomalies) of the event-based systems described using DERA are crucial. Any flaws in the design of a DERA application may lead to anomalies, for instance, deadlocks (when an application is blocked and no longer proceeds), and livelocks (when an application gets stuck in a never-ending loop).

As explained above, a snapshot of a DERA application comprises event actors, which are are "*conceptually*" connected to each other via events sent and received to accomplish a particular goal, for instance, orchestrating functions provided by a number of service platforms, processing a customer order, handling a travel itinerary request, and so forth. Thus, given such a snapshot, it is possible to leverage existing expressive and powerful formalisms for concurrent and distributed systems to analyze DERA-based application snapshots. Petri nets

(PNs) [7] have been chosen in our work for formalizing and analyzing the properties of DERA-based applications because they provide sufficient expressiveness for modeling the DERA constructs, and they are extensively studied and used in both academia and industry in various application domains [7]. As a result, we can benefit from numerous analysis techniques that have been developed for Petri nets [7] as well as a plethora of verification tools[4]. Nevertheless, other similar formalisms such as π-calculus [8] or communicating sequential processes [9] could have been used as well. Further details on Petri nets can be found [7]. Here we recall the basic definition of PNs.

Definition 2: A Petri net PN can be described by a tuple (P, T, F), where

- P is a finite set of places graphically represented as circles. The sets of input and output transitions of a place p are denoted by $\bullet p$ and $p\bullet$, respectively.
- T is a finite set of transitions graphically represented as boxes. The sets of input and output places of a transition t are denoted by $\bullet t$ and $t\bullet$, respectively.
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (relations) graphically represented as directed arrows.
- $P \cap T = \varnothing$ and $P \cup T \neq \varnothing$.

A state of a Petri net is described by a marking $\mu : P \to \mathbb{N}$ that assigns tokens to each place $p \in P$. The dynamic behavior of a Petri net is defined as following:

- A transition t is *enabled* in a marking μ iff $\forall p \in \bullet t : \mu(p) \geq |(p, t) \in F|$
- An enabled transition t can *fire* (or *execute*) such that a new marking μ' is achieved by the firing rule: $\forall p \in P : \mu'(p) = \mu(p) - |(p, t) \in F| + |(t, p)| \in F|$.

Our approach first introduces the descriptions of DERA constructs using Petri nets. Based on these descriptions, we can achieve a formal representation of a DERA snapshot at a certain point in time in terms of Petri nets. Such formal representation can be used to analyze important properties of the underlying DERA-based systems and identify various anomalies at design time and runtime.

B. Describing DERA constructs using Petri nets

The first step in our approach to verification of DERA snapshots is to enable the specification of DERA constructs using Petri nets. In Table II, we present in detail the formal mapping rules for describing DERA constructs in terms of Petri nets' elements such as *places*, *transitions*, and *arcs*. The first four rules are for specifying DERA event actors. Please note that the execution of an `EventActor` or a `Condition` will be triggered by one of its input events. Therefore, a guarded Petri net is placed in front of an `EventActor` or `Condition` to satisfy this rule. Regarding `Barrier` and `Trigger`, the rules are slightly different. A `Barrier` is triggered when all of its input events arrive, whilst a `Trigger` will automatically fire all of its output events. An `EventBridge` just forwards all of its incoming events. The last rule illustrates the representation of the "*connections*" between the event actor a having the output event e and a number of actors $b_i, i = 1..n$, who are waiting for the event e. Please note that these connections are not real but rather deducted from the interfaces of the aforementioned actors captured at a certain point in time. We can see that the Petri net representation of a DERA actor also exposes the places corresponding to the input and output events of the actor's interfaces. The grey box of each Petri net shown in Table II denotes the internal structure of the corresponding event actor.

Definition 3 (Mapping $\mathcal{M}$): A DERA application snapshot $\mathcal{S} = (A, E, E_{start}, E_{finish})$ captured at a certain point in time can be mapped to a Petri nets representation $\mathcal{M}(\mathcal{S}) = (P^{\mathcal{M}}, T^{\mathcal{M}}, F^{\mathcal{M}})$,where $P^{\mathcal{M}} = \bigcup_{x \in A} P_x^{\mathcal{M}}$, $T^{\mathcal{M}} = \bigcup_{x \in A} T_x^{\mathcal{M}}$, and $F^{\mathcal{M}} = \bigcup_{x \in A} F_x^{\mathcal{M}}$. The corresponding $P_x^{\mathcal{M}}$, $T_x^{\mathcal{M}}$, and $F_x^{\mathcal{M}}$ can be achieved by applying the rules in Table II.

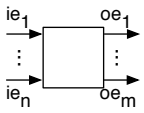
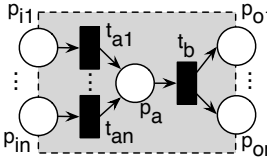
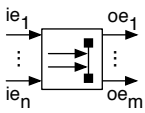
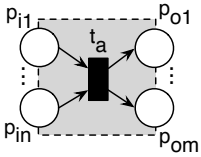
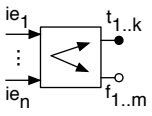
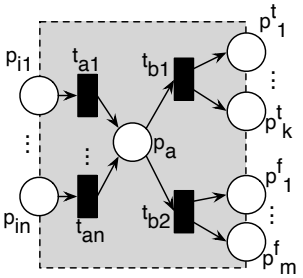
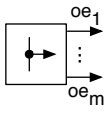
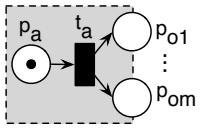
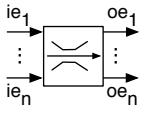
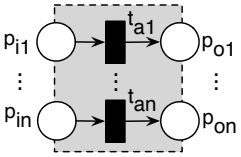
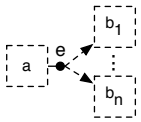
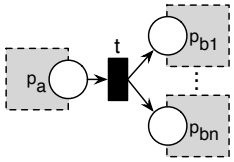
Among various types of PNs, free-choice Petri nets are special sub-class of Petri nets that can support the notions of concurrency and choice and exhibit a clear distinction between these notions. A free-choice Petri net (FCPN) is an ordinary Petri net $PN = (P, T, F)$ in which every arc from a place is either a unique outgoing arc or a unique incoming arc to a transition, i.e., $\forall p \in P, |p \bullet| \leq 1$ or $\bullet(p\bullet) = \{p\}$. Therefore, FCPNs are sufficient and appropriate to model DERA constructs and DERA snapshots. In addition, the significant advantage of using FCPNs is that the verification of some important properties such as liveness and boundedness can be established in polynomial time [7]. We will prove that the results of the mapping a DERA snapshot into Petri nets according to Definition 3 are FCPNs.

Lemma 1: Let $\mathcal{M}(\mathcal{S}) = (P^{\mathcal{M}}, T^{\mathcal{M}}, F^{\mathcal{M}})$ be the Petri nets representation of a DERA application snapshot $\mathcal{S}$ according to Definition 3. Then, $\mathcal{M}(\mathcal{S})$ is free-choice.

Proof. According to the mapping rules mentioned in Table II, the Petri nets representation of a DERA snapshot, except for the mapping of a `Condition`, consists of places that have only at most once outgoing arc connecting to a transition. That is, $|p\bullet| = 1$. A `Condition` is mapped to a Petri net that contains a place p_a with two outgoing arcs: (p_a, t_{b1}) goes to the transition t_{b1} (in case the associated `Predicate` is evaluated to `true`) and (p_a, t_{b2}) goes to the transition t_{b2} (in case the associated `Predicate` is evaluated to `false`). Thus, we need to prove that, $\bullet(p_a\bullet) = \{p_a\}$. Indeed, the

[4] An exhaustive list of Petri nets tools is available at http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html).

Table II
MAPPING OF DERA CONSTRUCTS TO PETRI NETS

DERA Construct (x)		Petri nets $\mathcal{M}(x)$	Mapping rules
EventActor			$\mathbf{P}_x^{\mathcal{M}}$ $\{p_{ii} = \mathcal{M}(ie_i) \vert i = 1..n, ie_i \in \bullet a\} \cup \{p_{oi} = \mathcal{M}(oe_i) \vert i = 1..m, oe_i \in a\bullet\} \cup \{p_a\}$ $\mathbf{T}_x^{\mathcal{M}}$ $\{t_{ai} \vert i = 1..n\} \cup \{t_b\}$ $\mathbf{F}_x^{\mathcal{M}}$ $\{(p_{ii}, t_{ai}) \vert i = 1..n\} \cup \{(t_{ai}, p_a) \vert i = 1..n\} \cup \{(p_a, t_b)\} \cup \{(t_b, p_{oi}) \vert i = 1..m\}$
Barrier			$\mathbf{P}_x^{\mathcal{M}}$ $\{p_{ii} = \mathcal{M}(ie_i) \vert i = 1..n, ie_i \in \bullet a\} \cup \{p_{oi} = \mathcal{M}(oe_i) \vert i = 1..m, oe_i \in a\bullet\}$ $\mathbf{T}_x^{\mathcal{M}}$ $\{t_a\}$ $\mathbf{F}_x^{\mathcal{M}}$ $\{(p_{ii}, t_a) \vert i = 1..n\} \cup \{(t_a, p_{oi}) \vert i = 1..m\}$
Condition			$\mathbf{P}_x^{\mathcal{M}}$ $\{p_{ii} = \mathcal{M}(ie_i) \vert i = 1..n, ie_i \in \bullet a\} \cup \{p_j^t = \mathcal{M}(t_i) \vert i = 1..k, t_i \in a\bullet_{true}\} \cup \{p_j^f = \mathcal{M}(f_i) \vert i = 1..m, f_i \in a\bullet_{false}\} \cup \{p_a\}$ $\mathbf{T}_x^{\mathcal{M}}$ $\{t_{ai} \vert i = 1..n\} \cup \{t_{b1}, t_{b2}\}$ $\mathbf{F}_x^{\mathcal{M}}$ $\{(p_{ii}, t_a) \vert i = 1..n\} \cup \{(t_{ai}, p_a) \vert i = 1..n\} \cup \{(p_a, t_{b1}), (p_a, t_{b2})\} \cup \{(t_{b1}, p_i^t) \vert i = 1..k\} \cup \{(t_{b2}, p_i^f) \vert i = 1..m\}$
Trigger			$\mathbf{P}_x^{\mathcal{M}}$ $\{p_a\} \cup \{p_{oi} = \mathcal{M}(oe_i) \vert i = 1..m, oe_i \in a\bullet\}$ $\mathbf{T}_x^{\mathcal{M}}$ $\{t_a\}$ $\mathbf{F}_x^{\mathcal{M}}$ $\{(p_a, t_a\} \cup \{(t_a, p_{oi}) \vert i = 1..m\}$
EventBridge			$\mathbf{P}_x^{\mathcal{M}}$ $\{p_{ii} = \mathcal{M}(ie_i) \vert i = 1..n, ie_i \in \bullet a\} \cup \{p_{oi} = \mathcal{M}(oe_i) \vert i = 1..n, oe_i \in a\bullet\}$ $\mathbf{T}_x^{\mathcal{M}}$ $\{t_{ai} \vert i = 1..n\}$ $\mathbf{F}_x^{\mathcal{M}}$ $\{(p_{ii}, t_{ai}) \vert i = 1..n\} \cup \{(t_{ai}, p_{oi}) \vert i = 1..n\}$
Relationships			$\mathbf{P}_x^{\mathcal{M}}$ $\{p_a = \mathcal{M}(e) \vert e \in a\bullet\} \cup \{p_{bi} = \mathcal{M}(e_i) \vert i = 1..n, e_i \in \bullet b\}$ $\mathbf{T}_x^{\mathcal{M}}$ $\{t\}$ $\mathbf{F}_x^{\mathcal{M}}$ $\{(p_a, t) \vert i = 1..n\} \cup \{(t, p_{bi}) \vert i = 1..n\}$

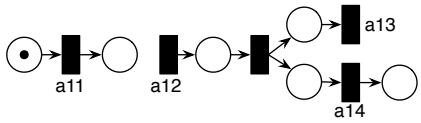
two transitions t_{b1} and t_{b2} belong to the Petri nets corresponding to the internal structure of a `Condition`. Hence, there are no other arcs from/to t_{b1} and t_{b2} except (p_a, t_{b1}) and (p_a, t_{b2}) (see Table II). The mapping rules yield $p_a\bullet = \{t_{b1}, t_{b2}\}$ and $\bullet t_{b1} = \{p_a\}$ and $\bullet t_{b2} = \{p_a\}$. That implies $\bullet(p_a\bullet) = \bullet(\{t_{b1}, t_{b2}\}) = \bullet t_{b1} \cup \bullet t_{b2} = \{p_a\}$. Therefore, the Petri net is free-choice.□

C. Petri nets-based analysis of DERA snapshots

The Petri nets representation $\mathcal{M}(\mathcal{S})$ of a DERA snapshot $\mathcal{S}$ captured either at design time or runtime will become inputs to verification tools for analyzing the application to ensure the absence of anomalies such as deadlocks and livelocks. In particular, we can support the verification of the properties such as "absence of dead event actors" and "proper completion" (i.e., no undesirable situations such as deadlocks or livelocks). In Figure 5, we illustrate some simple examples of DERA snapshots that can lead to such anomalies along with the corresponding Petri nets. For the sake of clarity, we have annotated the resulting Petri nets with event actors' names and reduced some redundant elements.

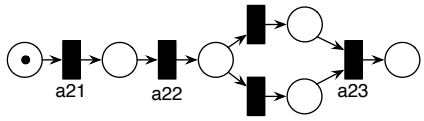
In Figure 5(a), two `EventActors` `a12` and `a13` are dead because they are either missing input or output events. As a result, `a12` can never be performed. The execution of `a13` likely leads to a deadlock because no subsequent firing and execution can happen after it

```
Trigger a11 output[e11]
EventActor a12 input[] output[e12]
EventActor a13 input[e12] output[]
EventActor a14 input[e12] output[e13]
```



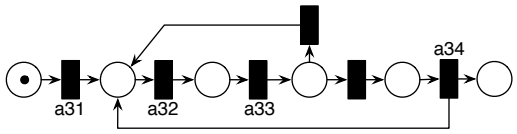
(a) Dead event actors (no input or output)

```
Trigger a21 output[e21]
Condition a22 input[e21] when-true[e22] when-false[e23]
Barrier a23 input[e22, e23] output[e24]
```



(b) Deadlocks (jamming before reaching the end)

```
Trigger a31 output[e31]
EventActor a32 input[e31] output[e32]
Condition a33 input[e32] when-true[e31] when-false[e33]
Barrier a34 input[e33] output[e31,e34]
```



(c) Livelocks (trapping in endless cycles)

Figure 5. Illustration of potential anomalies

finishes. Figure 5(b) illustrates a deadlock scenario in which the `Barrier a23` waits for two events `e22` and `e23` forever because only one of these events is fired by the `Condition a22`. The last scenario in Figure 5(c) demonstrates that a livelock situation may happen in two cases. Unless the associated `Predicate` of the `Condition a33` returns `false`, there will be a possibly unintentional never-ending loop. Even worse, we see that the `Barrier a34` will emit simultaneously two events `e31` and `e34`. The event `e31` is fed back to the `EventActor a32`, and therefore, triggers the execution of `a32`. As a consequence, this always yields endless execution cycles.

Please note that the aforementioned anomalies may occur due to unintended design and development mistakes of DERA-based applications. Therefore, the huge benefit of using formal specifications and verifications, e.g., Petri nets, is to help reducing and eliminating such anomalies early at design time or during the course of the execution of DERA applications. The verification can also be used for analyzing DERA applications before the deployment or actor substitutions at runtime [6].

D. Implementation and tool support

There are a diversity of existing tools that support the verification of Petri nets representations. Fortunately, many of these tools adopt the Petri Net Markup Language (PNML) defined by the standard ISO/IEC 15909[5]. PNML is an XML-based interchange format for Petri nets aiming at the openness and extensibility (i.e., allow everyone to extend the fundamental set of general features of all types of Petri nets and devise specific features of a concrete Petri net type). In this work, we introduce a template-based model transformation approach for automatically mapping DERA constructs and applications onto Petri nets according to the formal definition in Table II.

```
class DERA2PNML {
  val index = new AtomicInteger
  ...
  def dispatch map(Barrier barrier){
    val inputs = barrier.input
    val outputs = barrier.output
    '''
    <!-- Barrier "«barrier.name»" -->
    «FOR i : 1..inputs.size»
      <place id="«inputs.get(i-1).name»" />
      <arc id="arc_«index.andIncrement»" source="«inputs.get(i-1)
        .name»" target="barrier_«barrier.name»_ta" />
    «ENDFOR»
    <transition id="barrier_«barrier.name»_ta" />
    «FOR i : 1..outputs.size»
      <place id="«outputs.get(i-1).name»" />
      <arc id="arc_«index.andIncrement»" source="barrier_«barrier
        .name»_ta" target="«outputs.get(i-1).name»" />
    «ENDFOR»
    '''
  }
  def dispatch map(EventActor actor){
    val inputs = actor.input
    val outputs = actor.output
    '''
    <!-- EventActor "«actor.name»" -->
    «FOR i : 1..inputs.size»
      <place id="«inputs.get(i-1).name»" />
      <transition id="«actor.name»_ta_«i»" />
      <arc id="arc_«index.andIncrement»" source="«inputs.get(i-1)
        .name»" target="«actor.name»_ta_«i»" />
      <arc id="arc_«index.andIncrement»" source="«actor.name»_ta_
        «i»" target="actor_«actor.name»_pa" />
    «ENDFOR»
    <place id="actor_«actor.name»_pa" />
    <transition id="actor_«actor.name»_tb" />
    <arc id="actor_«actor.name»_arc_«index.andIncrement»" source=
      "actor_«actor.name»_pa" target="actor_«actor.name»_tb" />
    «FOR i : 1..outputs.size»
      <place id="«outputs.get(i-1).name»" />
      <arc id="arc_«index.andIncrement»" source="actor_«actor.
        name»_tb" target="«outputs.get(i-1).name»" />
    «ENDFOR»
    '''
  }
  def dispatch map(Condition condition){
    val inputs = condition.input
    val trueOutputs = condition.^true
    val falseOutputs = condition.^false
    '''
    <!-- Condition "«condition.name»" -->
    «FOR i : 1..inputs.size»
      <place id="«inputs.get(i-1).name»" />
      <transition id="«condition.name»_ta_«i»" />
      <arc id="arc_«index.andIncrement»" source="«inputs.get(i-1)
        .name»" target="«condition.name»_ta_«i»" />
      <arc id="arc_«index.andIncrement»" source="«condition.name»
        _ta_«i»" target="condition_«condition.name»_pa" />
    «ENDFOR»
    <place id="condition_«condition.name»_pa" />
    <transition id="condition_«condition.name»_tb1" />
```

[5]http://www.iso.org/iso/catalogue_detail.htm?csnumber=43538

```
    <transition id="condition_«condition.name»_tb2" />
    «FOR i : 1..trueOutputs.size»
      <place id="«trueOutputs.get(i-1).name»" />
      <arc id="arc_«index.andIncrement»" source="condition_«
       condition.name»_tb1" target="«trueOutputs.get(i-1).name»"
       />
    «ENDFOR»
    «FOR i : 1..falseOutputs.size»
      <place id="«falseOutputs.get(i-1).name»" />
      <arc id="arc_«index.andIncrement»" source="condition_«
       condition.name»_tb2" target="«falseOutputs.get(i-1).name»"
        />
    «ENDFOR»
    '''
  }
  ...
}
```

Listing 2. An excerpt of the code for mapping DERA to PNML

The DeraDSL used in the previous example has been implemented using the Eclipse Xtext DSL framework[6]. The DeraDSL editor generated by Xtext can support several powerful features such as syntax highlighting, content assist and auto-completion, validation and quick fixes, automated external cross-references resolutions, and so on. The transformation of DERA constructs to PNML is implemented using Xtend[7], a statically-typed language built on top of Java provided by the Xtext framework. We present in Listing 2 an excerpt of the Xtend code for mapping basic DERA constructs such as `Barrier`, `Condition`, and `EventActor` onto PNML, respectively. The mapping of other actors can be achieved in the same way. Please note that the generation templates are defined inside the pair of three apostrophes (i.e., "'...'").

We illustrate in Table III the transformation of an excerpt of the warehouse operator defined in Listing 1 onto the corresponding PNML. The resulting PNML description can be used in existing Petri nets-based tools for analysis and verification.

For instance, we can use the tool Workflow Petri net Designer (WoPeD) 3.0.2[8], to import the PNML description that is automatically generated from a snapshot of the warehouse operator (see Figure 1 and Listing 1). The graphical view of the Petri nets representation is shown in the left-hand side of Figure 6 and the analysis of properties of the Petri nets representation such as well-structuredness, boundedness, and liveness, are shown in the right-hand side, respectively.

V. Related work

The advantages of event-driven architectures have been extensively studied in different areas such as middleware infrastructure [2], event-based coordination [3], active database systems [4], workflow and business process management [10], and service-oriented architectures [5]. Event-condition-action (ECA) rules – proposed in active database systems [11] – are a declarative approach using rules for enabling reactions as a certain event occurs and particular conditions hold. A number of studies have been conducted aiming at using ECA rules for executing business processes [12]. However, most of the ECA-based approaches have limitations with regard to their applicability as it is tedious to generate or extract, manage, and analyze ECA rules for complex processes [13]. Our approach, in contrast, offers easy-to-understand graphical notations and formalisms for the stakeholders and an automated mapping from the graphical notations onto the formal representations.

There are a considerable amount of studies focusing on formalizing and supporting model checking of event-based systems. In his dissertation work, Madl presented an approach that defines a semantic domain based on timed automata and discrete event systems to support

Table III
Example of translating DERA constructs to PNML

DERA

```
EventActor confirmStorageLocations input [eInquiried] output
  [eConfirmed]
Barrier gate input [eConfirmed, eVideoReceived] output [
 eSynched]
Condition c1 input [eSynched] when-true [eEnoughLocations]
 when-false [eNotEnoughLocations]
```

PNML

```
<!-- EventActor "confirmStorageLocations" -->
<place id="eInquiried" />
<transition id="confirmStorageLocations_ta_1" />
<arc id="arc_0" source="eInquiried" target="
 confirmStorageLocations_ta_1" />
<arc id="arc_1" source="confirmStorageLocations_ta_1" target
 ="actor_confirmStorageLocations_pa" />
<place id="actor_confirmStorageLocations_pa" />
<transition id="actor_confirmStorageLocations_tb" />
<arc id="actor_confirmStorageLocations_arc_2" source="
 actor_confirmStorageLocations_pa" target="
 actor_confirmStorageLocations_tb" />
<place id="eConfirmed" />
<arc id="arc_3" source="actor_confirmStorageLocations_tb"
 target="eConfirmed" />
<!-- Barrier "gate" -->
<place id="eConfirmed" />
<arc id="arc_4" source="eConfirmed" target="barrier_gate_ta"
  />
<place id="eVideoReceived" />
<arc id="arc_5" source="eVideoReceived" target="
 barrier_gate_ta" />
<transition id="barrier_gate_ta" />
<place id="eSynched" />
<arc id="arc_6" source="barrier_gate_ta" target="eSynched"
 />
<!-- Condition "c1" -->
<place id="eSynched" />
<transition id="c1_ta_1" />
<arc id="arc_7" source="eSynched" target="c1_ta_1" />
<arc id="arc_8" source="c1_ta_1" target="condition_c1_pa" />
<place id="condition_c1_pa" />
<transition id="condition_c1_tb1" />
<transition id="condition_c1_tb2" />
<place id="eEnoughLocations" />
<arc id="arc_9" source="condition_c1_tb1" target="
 eEnoughLocations" />
<place id="eNotEnoughLocations" />
<arc id="arc_10" source="condition_c1_tb2" target="
 eNotEnoughLocations" />
```

[6]http://www.eclipse.org/Xtext
[7]http://www.eclipse.org/xtend
[8]http://www.woped.org

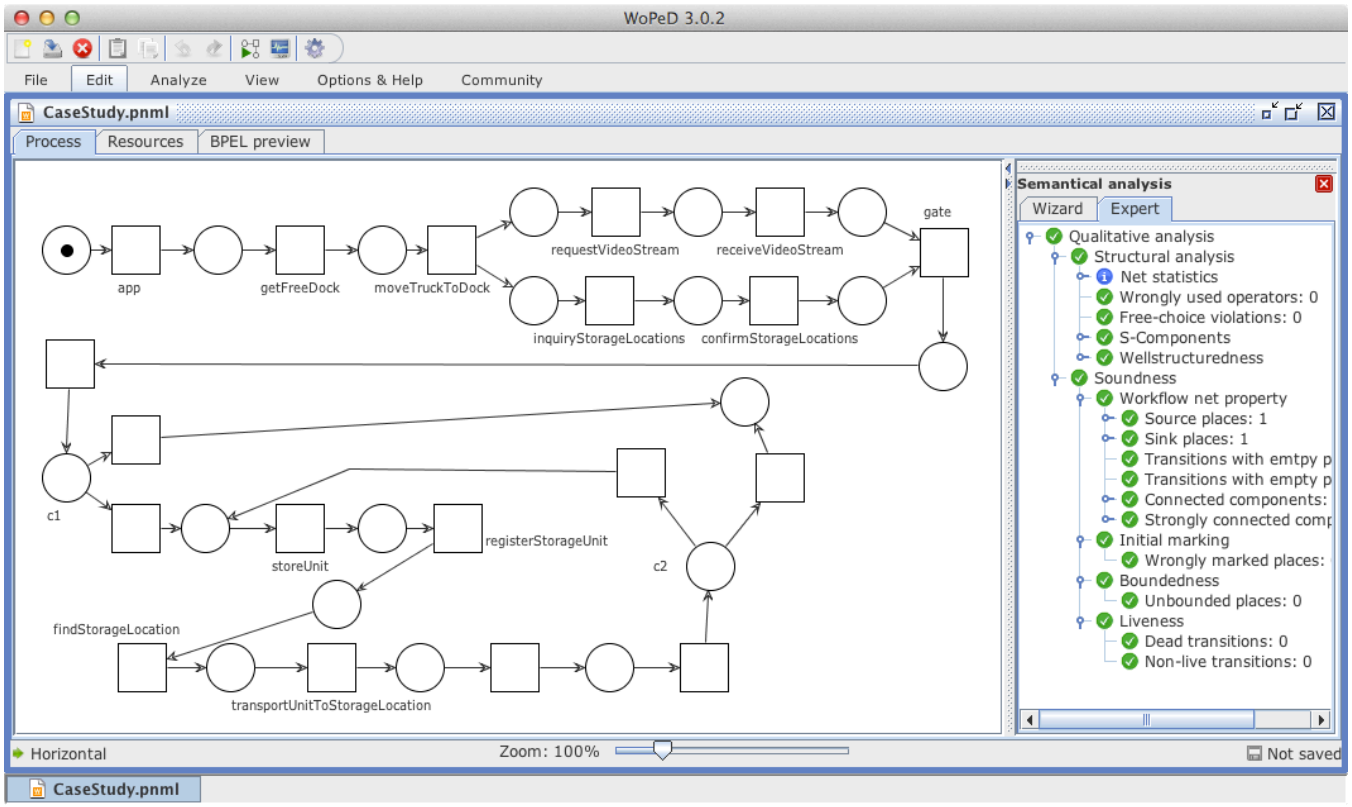


Figure 6. Petri nets-based analysis of a snapshot of the warehouse operator

the formalization and verification of distributed real-time embedded systems [14]. Madl's approach rather focuses on low-level and fine-grained concepts such as machines, execution threads, tasks, schedulers, and buffers whilst our notions and formalism presented in this paper are of high-level of abstraction and rather closer to the developers' perception. Another approach proposed by Atlee and Gannon [15] concentrates on formalizing event-driven system requirements based on computational tree logic (CTL). Our approach can be extended to be able to verify the conformance of event-based systems design and implementation using DERA constructs based on formal requirement specifications achieved by using this approach.

There are a rich body of existing work on specification and verification of distributed event-based systems based on temporal logic [16], trace semantics [17], algebraic semantics [18], ontology [19], π-calculus [20], and operational semantics [21]. To the best of our knowledge, the aforementioned approaches mainly target the publish/subscribe messaging patterns. The dynamic event actors-based framework that we consider in this paper relies on asynchronous communication style to realize the event channels, and therefore, can benefit from the formal foundations resulting in these approaches. Unfortunately, none of these approaches aims at introducing appropriate representations and formalisms of event-based systems that are close to the developers' perception. Moreover, our approach also provide an automated mapping between the system's graphical and formal representations.

The theoretical foundation of our work benefits from existing literature on Petri nets, especially on the formalization of concurrency and choice constructs [7]. Nevertheless, other formalisms such as π-calculus [22] and Communicating Sequential Processes (CSP) [9] strongly support these constructs, and therefore, are also applicable in our approach. Petri nets has been chosen as the underlying formalism in our approach because of the intuitive graphical representation backed by powerful formalization along with a plethora of existing commercial and open-source tools support.

VI. Conclusion

We present in this paper a novel approach that leverages graphical representations of event-based systems with reduced non-determinism that are closer to the developers' perception. In particular, the dynamic event actors framework (DERA) is used for describing the underlying event-based systems. The snapshots of a DERA-based description obtained at design time or

runtime shall be automatically translated into corresponding Petri net representations. Formal analysis and verification of the event-based systems under consideration are performed based on the resulting Petri nets. We developed a proof-of-concept implementation of our approach and showed the applicability of our approach through an industrial case study. Our follow-on endeavors aim at further investigating relevant open issues not covered yet in the scope of this paper. Among the open topics is to reduce the complexity of analyzing Petri nets of the event actors-based application spanning over multiple execution domains. Another potential task is to enable the capability of intuitively monitoring and debugging event-based systems based on the captured snapshot and the corresponding graphical representations.

Acknowledgment

This work was partially supported by the EU's Seventh Framework Programme Project INDENICA (http://www.indenica.eu), Grant No. 257483 and the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001.

References

[1] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed Event-Based Systems*, 1st ed. Springer, 2006.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Trans Comput Syst*, vol. 19, no. 3, pp. 332–383, Aug. 2001.

[3] F. Arbab and C. L. Talcott, Eds., *5th Int'l Conf. Coordination Models and Languages*, ser. LNCS, vol. 2315. Springer, 2002.

[4] N. W. Paton and O. Díaz, "Active database systems," *ACM Comput. Surv.*, vol. 31, no. 1, pp. 63–103, Mar. 1999.

[5] S. Ganesan, Y. Yoon, and H.-A. Jacobsen, "NIÑOS take five: the management infrastructure for distributed event-driven workflows," in *5th ACM Int'l Conf. on Distributed event-based system (DEBS)*. ACM, 2011, pp. 195–206.

[6] H. Tran and U. Zdun, "Event-driven actors for supporting flexibility and scalability in service-based integration architecture," in *20th Int'l Conf. on Cooperative Information Systems (CoopIS)*. Springer Heidelberg, 2012, pp. 164–181.

[7] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[8] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Information and Computation*, vol. 100, no. 1, pp. 1–40, Sep. 1992.

[9] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, Apr. 1985.

[10] W. M. P. van der Aalst, "Formalization and verification of event-driven process chains," *Inform. Software Tech.*, vol. 41, no. 10, pp. 639–650, Jul. 1999.

[11] N. W. Paton, *Active Rules in Database Systems (Monographs in Computer Science)*. Springer-Verlag New York, Inc., 1998.

[12] G. Kappel, S. Rausch-Schott, and W. Retschitzegger, "A framework for workflow management systems based on objects, rules and roles," *ACM Comput. Surv.*, vol. 32, no. 1es, pp. 27–es, Mar. 2000.

[13] J. Bae, H. Bae, S. ho Kang, and Y. Kim, "Automatic control of workflow processes using ECA rules," *IEEE T. Knowl. Data En.*, vol. 16, no. 8, pp. 1010–1023, Aug. 2004.

[14] G. Madl, "Model-based Analysis of Event-driven Distributed Real-time Embedded Systems," PhD Thesis, University of California, Irvine, 2009.

[15] J. M. Atlee and J. Gannon, "State-based model checking of event-driven system requirements," *IEEE Trans. Software Eng.*, vol. 19, pp. 24–40, 1993.

[16] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," in *9th European Softw. Eng. Conference/11th ACM SIGSOFT Int'l Symposium on Foundations of Softw. Eng.* ACM, 2003, pp. 257–266.

[17] L. Fiege, G. Mühl, and F. C. Gärtner, "Modular event-based systems," *The Knowledge Engineering Review*, vol. 17, no. 4, pp. 359–388, Dec. 2002.

[18] J. L. Fiadeiro and A. Lopes, "An algebraic semantics of event-based architectures," *Mathematical. Structures in Comp. Sci.*, vol. 17, no. 5, pp. 1029–1073, Oct. 2007.

[19] A. Scherp, T. Franz, C. Saathoff, and S. Staab, "F–a model of events based on the foundational ontology DOLCE+DnS ultralight," in *5th Int'l Conf. on Knowledge Capture (K-CAP)*. ACM, 2009, pp. 137–144.

[20] E. Posse and J. Dingel, "Kiltera: A language for timed, event-driven, mobile and distributed simulation," in *IEEE/ACM 14th Int'l Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, oct. 2010, pp. 87–96.

[21] J.-P. Müller, "Towards a formal semantics of event-based multi-agent simulations," in *International Workshop on Multi-Agent-Based Simulation IX (MABS)*, 2008, pp. 110–126.

[22] R. Milner, *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, Jun. 1999.