Towards Specification Completion for Systems with Emergent Behavior based on DevOps

Mohamed Toufik Ailane, Adina Aniculaesei, Christoph Knieke, Andreas Rausch, Fauzi Sholichin
Clausthal University of Technology
Institute for Software and Systems Engineering
Clausthal-Zellerfeld, Germany
Email: mohamed.toufik.ailane@tu-clausthal.de, adina.aniculaesei@tu-clausthal.de, christoph.knieke@tu-clausthal.de, andreas.rausch@tu-clausthal.de, fauzi.sholichin@tu-clausthal.de

***Abstract*—Software systems may experience multiple emergent behaviors during their operation time. These emergent system behaviors occur when system engineers develop their system under the closed-world assumption, but this assumption is not met during its operation. This means that system engineers work on the basis that they have complete knowledge of the system and its environment during its design, when the system specification that has been created is actually incomplete. In this paper, an observation of an emergent behavior is considered to be a solid proof that the system model specification is still incomplete. A conceptual framework is proposed to harness the emergent behavior and complete the system specification that is provided during the its design. The framework consists of two parts, system development and system operations. It is built on a model-driven approach in order to provide a clear definition of the emergent behavior and a concrete development scheme. The framework exploits the DevOps paradigm as a successful paradigm to achieve the ultimate goal of developing complete system models through the *continuous specification completion* based on the observed emergent behavior. The goal of this framework is to help develop high-quality and reliable emergent systems based on the specification derived from the emergent behavior that occurred at run time.**

***Index Terms*—DevOps, Software evolution, Complex systems, Formal methods, Software development process.**

I. Introduction

In the last decade, Model-Driven Engineering (MDE) has been a mature and successful approach for the development of software systems [1]. Similar to other scientific fields, in which the use of models facilitates greatly the understanding of the phenomena under analysis, the usage of models in software engineering contributes to a better understanding of the system under development. The core of the model-driven engineering is based on using models as abstract representations of different aspects of the system under development, e.g., system structure or system behavior.

The essential benefit of system models is that they represent abstractions of certain aspects of the system under analysis, that leave out certain system features that are irrelevant for the analysis from a the given perspective (cf. [1]). Since system models are built with goals in mind, they can be considered *good/bad* or *correct/not correct* only with respect to these goals (cf. [2]). These goals are often refined into concrete system requirements during the process of requirements elicitation and analysis. Thus, checking whether the system model is correct with respect to a certain goal means checking whether the model is correct with regard to the respective system requirements. Formal verification methods can be used to verify system models with respect to the specified system requirements. In order to apply formal verification methods, the system model and the system requirements must be made verifiable. This means that both the system model and the system requirements need to be formalized using specific modeling and specification formalism, e.g., finite-state machines for the formal description of the system model and linear temporal logic (LTL) for the formal specification of the system requirements. Having verifiable models enables the development of what is referred to as *correct-by-construction* systems. Correct-by-construction system development is important and necessary particularly when designing and developing safety-critical systems. However, building a verifiable system model that represents the right level of abstraction for the system under analysis is not a trivial task. In fact, such a task is costly in terms of time and resources. Since it advocates for quick delivery of a functional, albeit incomplete system, agile development is an approach that can help curve the costs of system development (cf. [3]).

Agile development is a paradigm that has been widely adopted since it has been introduced two decades ago through the *Agile Manifesto* (cf. [3]). This document introduces a new way of thinking when it comes to software development. In contrast to traditional development processes, e.g., the V-model, in which the software system is delivered only after it passes all acceptance tests, the school of thought of agile development encourages system engineers to incorporate the customer's feedback in incremental development sprints in order to periodically deliver new and functional versions of the software system. On one side, the incremental development reduces time to market, and on the other side, it contributes to an increased customer satisfaction due to the involvement of the customer in the development process. Different processes are proposed under the umbrella of the Agile Manifesto, among them also DevOps. As its name indicates, DevOps is an agile approach that attempts to fill the gap between development (Dev) and operations (Ops) (cf. [4]). This gap is filled by automating different processes that support continuous deployment and continuous delivery of the system under development. Nevertheless, the DevOps paradigm faces multiple challenges, starting from a unifying definition of DevOps and adequate evaluation metrics, and continuing with the choice of tool set necessary to enable a real collaboration between the development team and the operations team (cf. [5]). In this paper, we address one important challenge of DevOps. This challenge is often referred to in literature as *closing the DevOps loop*. Its focus lies on the feedback phase within the DevOps loops and how to use this feedback to further improve the quality of the software system under development.

In the DevOps paradigm, the requirements catalogue as well as the architecture models and the implementation are considered to be incomplete. Incomplete development models are the main source of emergent behavior during the system operation (cf. [6]). In this paper we propose a conceptual framework which combines MDE and DevOps in a hybrid approach in order to harness the system emergent behavior that results from the incomplete specification of the system development models. Our framework relies on a model-driven engineering approach throughout the development phases of the DevOps process. In order to formalize the system behavior observed at run-time, our framework enhances the DevOps tool set with specification mining tools. Since the development models and the mined models are formalized (or semi-formalized), our framework provides a systematic approach for closing the DevOps loop. If the observed system behavior is an emergent behavior, the automation tools in our framework are used for the completion of the development models specification using the specification extracted from the observed emergent behavior.

The remainder of this paper is organized as follows. Section II discusses some related work to the proposed approach. Section III provides the necessary fundamentals and foundations to understand and establish the contributed framework. Section IV contains the details of the main contribution of this work. We offer a simple scenario in Section V in order to demonstrate how can such a conceptual framework be adopted. We conclude out work and reflect on possible future research directions in Section VI.

II. Related Work

In this section, we review several related works that discuss MDE and the adoption of the DevOps principle to support the continuous development of system models. To the best of our knowledge, there exists no other approach that considers exploiting the emergent behavior in a DevOps process in order to complete the specification of the system under development. Therefore, we relate to the research work carried out on the integration of model-driven development approaches in the DevOps development process. Benoit et al. propose their vision for supporting model-driven DevOps practice and explain the challenge of moving from *Dev* to *Ops* and vice versa (cf. [7]). In order to move from development to operation process, dedicated support to test and deploy automatically is required for further activities in this direction (cf. [7]). Furthermore, moving from operation to development needs descriptive run-time models. These models can be learned during the monitoring process and then linked back to the design models.

Hugues et al. propose ModDevOps as a combination of model-based software engineering (MBSE) and DevSecOps (cf. [8]). DevSecOps is an iteration of DevOps with wrapped security as an additional layer to the continual development and operation process (cf. [9]). The ModDevOps is a systems/software co-engineering practice that has as a goal the unification of systems engineering (*Mod*), software development (*Dev*), and software operation (*Ops*) (cf. [8]. The first step, *Mod*, consists of planning the system requirements, modeling the system architecture, and defining the interaction points between the different models through virtual integration. The second step, *Dev*, is the software implementation and assembling the multiple components in the whole software system. The third step, *Ops*, contains deployment and executes of the built software either in the target operational environment or in a simulation, monitoring and data analysis. Following a typical DevOps cycle, the authors addressed two main challenges that apply to the first half of the process, Dev-to-Ops. These challenges are the integration of model-driven techniques to DevOps and integration of heterogeneous artifacts (cf. [7]). Based on these two challenges, Hugues et al. propose TwinOps which extends of ModDevOps by combining it with the concept of digital twins (cf. [9]). TwinOps is based on a single central concept: leveraging engineering models from other domains (mechanics, electronics, etc.) to evaluate software-intensive systems against accurate representations of the environment (cf. [9]).

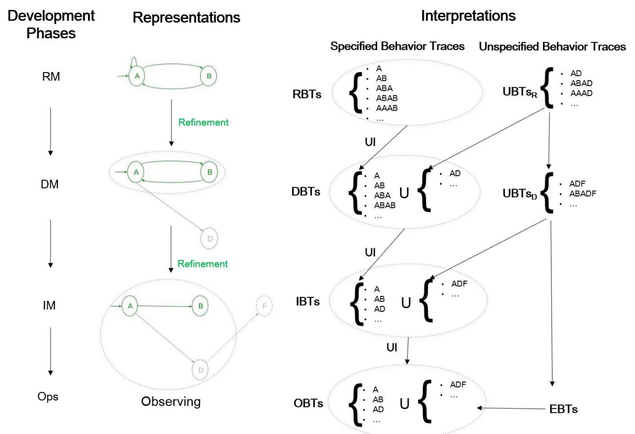


Fig. 1. Extended Abstraction Refinement Model (EARM) [6]

Flexible methods are needed to evaluate systems based on the ever-changing requirements of the two Dev-Ops dimensions (cf. [7], [10]). Thus, the seamless blending of the Dev-to-Ops and Ops-to-Dev continuum will provide cutting-edge features to enhance the modeling process.

III. Theoretical Background

A. Extended Abstraction Refinement Model

In our previous work [6], we proposed the extended abstraction refinement model (extended ARM) to include the concept of the emergent behavior, which does not appear in the standard abstraction refinement model (ARM). The abstraction refinement model has been first introduced by Keller [11] and is a model of the software evolution process [6]. Emergent behavior is behavior that is not specified during system development, instead it is observed during system operation [6]. The extended ARM shown in Figure 1 matches the development phases of the system life-cycle process - requirements analysis, system design, system implementation, and system operation - with the respective representation model of the software system in each of these different phases. Thus, the system development phases of requirements analysis, system design, and system implementation are matched with a requirement model (RM), a design model (DM), an respectively and implementation model (IM). For the system operation phase, there is a set of observations that are made during system operation [6]. Semantically, each of models RM, DM and IM is interpreted as a set of behavior traces. The extended ARM differentiates between specified and unspecified behavior traces. The specified behavior traces represent the interpretation of specified behavior in the requirements model, that is later refined in the behavior traces corresponding to the design model and the implementation model. The unspecified behavior traces are considered to be traces that can exist in the design and in the implementation of the software system, without being specified in the requirements analysis phase [6]. Emergent system behavior consists of unspecified system behavior observed during the system operation.

B. Specification Mining

Specification mining is a technique by which system properties can be inferred from a software system in an automated way or a semi-automated way. There are roughly two categories of system specification that can be mined for, namely specification formulated in temporal logic formula (cf. [7], [12]–[14]) and system specification described with the help of a finite-state models, e.g., finite state automata (FSA) (cf. [12], [15]–[17]). Both categories are capable of describing the dynamic actions of complete systems based on a collection of execution traces.

One of the tools used by researchers and practitioners for specification mining is Texada [13] [18]. The tool implements mining techniques for LTL specifications and allows the extraction of LTL specifications of arbitrary length. In addition, Texada takes the user-defined LTL property template and the execution traces as input and outputs a group of property instances for the given LTL property template [17]. Moreover, Texada also supports properties with two control thresholds, namely confidence threshold and support threshold. The confidence threshold permits the degree to which a mining property can be violated to be controlled by the user. The support threshold enables the user to decide how many times a property must be verified.

IV. Conceptual Framework for Specification Completion Based on the Observed Emergent Behavior

The conceptual framework proposed in this section enables the use of the emergent behavior observed at run-time and leverages exiting as well as specifically developed tools in order to complete the specification of the different models created during the system development phases, and thus close the DevOps loop. As shown in Figure 2, the observed behavior traces ($OBTs$) are compared against the development models in order to distinguish between the specified behavior refined from the previous development phases and the emergent behavior observed during system operation. Once the emergent behavior traces ($EBTs$) are recognized, the challenge is how to exploit these traces in order to close the DevOps loop.

A. From Development to Operations (Dev2Ops)

The system development follows a sequential development process, e.g., the waterfall model, from which four phases are taken into consideration: *requirements analysis*, *system design*, *system implementation*, and *system operation*. In the first three phases, the result of each development phase is a development model, that is built in a specific representation based on the tool used to create the model during the respective development phase, e.g., textual document for the requirements model, or UML diagram for the design model. Each model can be interpreted into a set of behavior traces. These traces can be used to check the refinement relation between the models created in the different phases of the development process or check if an emergent behavior is observed during the system operation.

Refinement is an activity which occurs in the different phases of the system development phases. In each phase, it consists of deriving a model from another model built in a previous phase of the development process, e.g., a design model may be derived in the system design phase from a requirements model that has been created in the requirements analysis phase. The derived model can describe behavior that was not specified in the model of the previous development phase. This is the case for example due to design constraints, design principles, coding platform conditions, or insufficient knowledge of the system designers during the previous development phase. The unspecified system behavior becomes the source of emergent system behavior, if this behavior is observed during system operation.

For systems built with the *"correct-by-construction"* paradigm in mind, the issue of the emergent behavior is considered to be detrimental and therefore it is handled by allocating more resources in order to guarantee that the refinement activity is applied to complete models and produces complete models. Hence, the manifestation of an emergent behavior at run-time is almost guaranteed not to take place. However, the cost for this guarantee comes sometimes at a great expense in terms of resources that are limited, e.g., time and work effort.

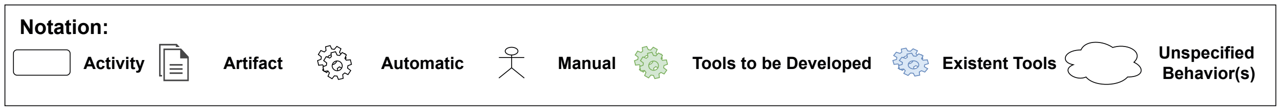
In contrast, the DevOps paradigm works on the principle of quick delivery of a functional system even if this may be built based on an incomplete requirements catalogue or an incomplete design model. The idea is to gather customer feedback as soon as possible, which is then used in the following development iteration to improve the system design and its implementation. The DevOps paradigm brings teams from system development and system operation at the same table, in order to understand the software system under development from all angles. It also gives the operators the chance to contribute to the system development through their feedback. The end goal is to produce a software system that complies with the changing requirements of all stakeholders and gives a better operation experience for customers that operate the system. Nevertheless, cyber-physical systems used in safety-critical applications may not tolerate such a paradigm, since any emergent behavior can potentially lead to disastrous consequences.

The conceptual framework in Figure 2 proposed a hybrid approach for the engineering of software systems. Such a system is considered an *emergent system*, because the observed emergent behavior is used in order to improve the specification of the system design models and further develop the system. Although the system is partly developed straightforwardly, the other part consists of emergent specifications and properties that can be extracted from the observed run-time system behavior. A fundamental aspect in the engineering of emergent systems is a definition of good and concrete techniques for the exploitation of the emergent system behavior that is observed during system operation.

B. From Operations to Development (Ops2Dev) - Closing the DevOps Loop

The emergent system behavior observed during system operation is used to complete the development models created during the development phases of the system, thus closing the DevOps loop. To demonstrate this, the waterfall model is adopted as system development process. Three major steps are carried out for the completion of the system specification: (1) requirements specification completion, (2) design specification completion, and (3) test case generation from the emergent system behavior.

1) Requirement Specification Completion: The requirement analysis phase usually results in a document that contains the intentions and goals of the different stakeholders for the system. A requirements engineering team is then responsible for the collection and processing of stakeholders' requirements in order to produce a consistent requirements model for further development phases. This model can be in the form of a textual document (informal), or it can be formalized into a formal representation. In order to enable a systematic and automatic specification completion, a formal representation is considered more useful than a textual document. At this level, the assumption is that the requirement model might be incomplete. This assumption comes from the fact that the developed system is emergent by definition, and thus, during system operation, it exhibits emergent behavior that corresponds to some system properties unspecified in the requirements model. The core principle of DevOps is the quick delivery of a functional system, albeit incomplete, in order to make use of the collected



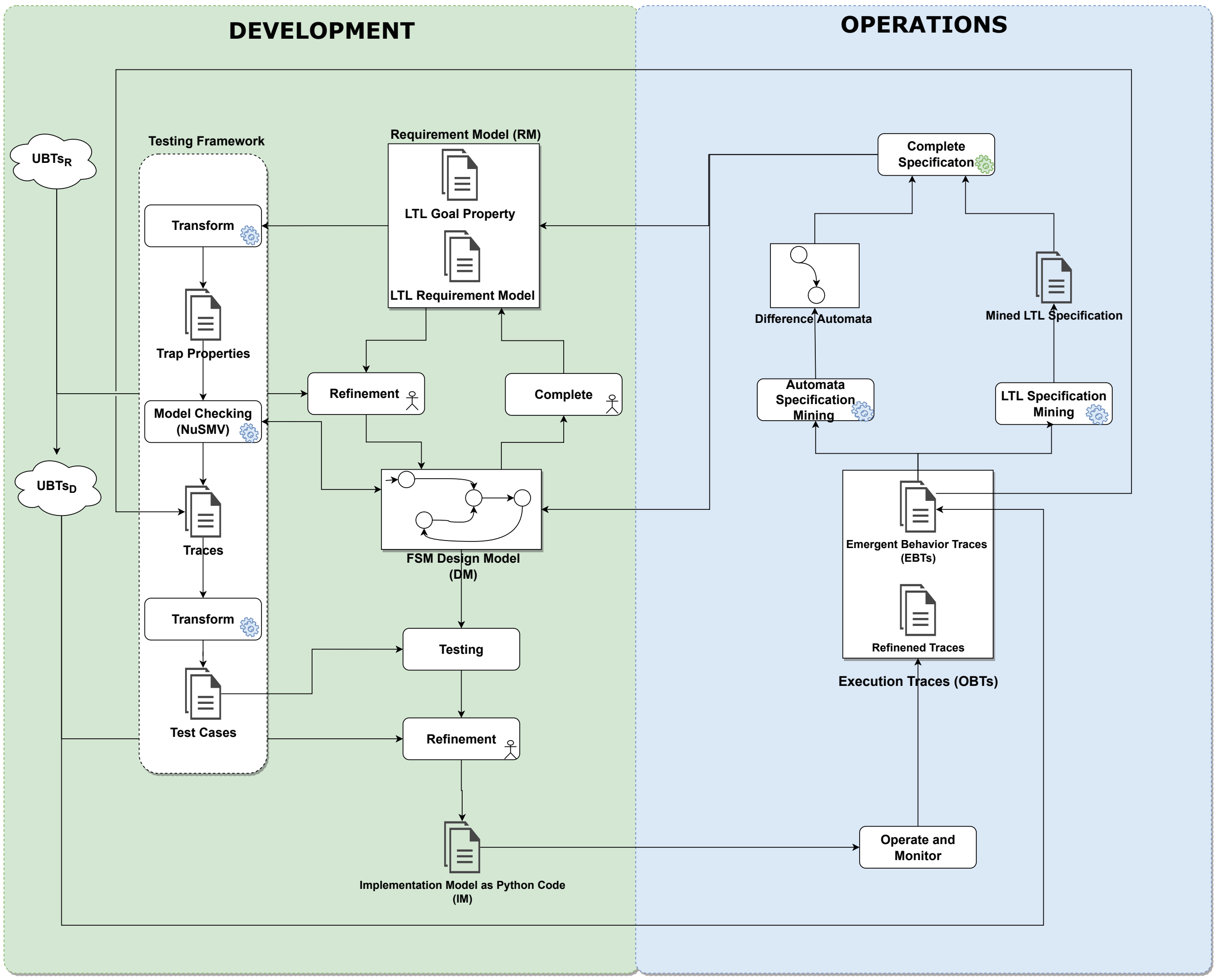


Fig. 2. Conceptual Framework for Software Specification Completion Based on the Extended ARM and DevOps.

customer feedback. This paradigm of development is realized through minimal specification of the models, with the expectation to complete the models in the next iteration of the DevOps loop. In case system specification is lacking at this level, the emergent behavior can be formalized on the operations side in the same formalism as the requirement model in order to reason about the lacking specifications that led to observing the emergent behavior in the first place. The emergent behavior should be harnessed to complete the specification of the requirement model either by: (1) defining and adding the missing specification so that the emergent behavior is admitted as part of the system and no longer regarded as emergent, or (2) adjusting the requirements model to block the emergent behavior in future iterations of the DevOps loop.

2) Design Specification Completion: System design activities produce a variety of models that cover different system aspects and views, e.g., structural view or behavioral view of the system. The design model(s) are the result of the refinement applied to the requirements model(s). Given RBT and DBT as two set of behavior traces resulting from the interpretation of the requirement model(s) and respectively of the design model(s), the refinement activity means guaranteeing that $DBT \subseteq RBT$. Since the requirements model(s) might be incomplete, it follows that design model(s) derived from the requirements model(s) through refinement may also be incomplete (cf. [6]). Moreover, the design model(s) might include further specification that has not been specified in the requirements model(s), e.g., due to design constraints or incomplete knowledge of the system designers. Model inference techniques can be exploited in this direction in order to achieve completion of the design model specification. An approach that is based on finite state machine mining is being prepared as part of future work.

3) Test Cases Generation based on Emergent Behavior Traces: Emergent behavior traces can be used as a basis to construct relevant test cases to check the developed system. Test cases can be constructed by test engineers either manually or using tool support which helps generate the test cases. In this paper, we extend the testing framework proposed in [19] to include the emergent behavior in the testing approach and integrate the extended testing framework into our conceptual framework.

The testing framework proposed in [19] consists of the automatic generation of the test cases based on the requirements model (cf. Figure 1 in [19]). First, the requirement model is formalized in the form of LTL properties, also called LTL obligations. The LTL obligations are developed manually and are refined in order to produce the design model. Automatic generation of requirements-based test cases relies on the core principle of model checking, i.e., whenever an error is found in the design model, the model checker provides a counterexample which shows the trace of the violated LTL property from the initial state to the error state in the design model (cf. [19]). Thus, for the purpose of the testing approach, *trap properties* are generated through the negation of the requirements model using a developed tool and specific requirements coverage criteria. The counterexamples obtained by verifying the system design model against the trap properties show how the original LTL specification is satisfied. The traces of these counterexamples are transformed into test cases via a developed software tool. The resulting test suite is executed on a set of system mutants and the results are evaluated with respect to a set of predefined test coverage criteria (cf. [19]).

To integrate the testing framework from [19] in this paper's conceptual framework, some differences with respect to the original work in [19] need to be considered. Firstly, the testing framework cannot be used as described in the original paper, because its starting premise is different from that of the DevOps paradigm. Aniculaesei et al. consider that the system operation begins only after the requirements models, the design models and the implementation models of the system under development are complete (cf. [19]). By comparison, in the DevOps paradigm a functional system is deployed and put into operation even though its requirements models and design models might be incomplete. Secondly, the trap properties are not negations of already specified system requirements. Instead, these properties represent unspecified behavior traces of the requirements model and of the design model. The counterexample trace obtained via model checking is then an emergent behavior trace that is observed during system operation.

V. Case Study: Agent-Based Systems

A. Scenario Description

The usage of the conceptual framework proposed in this paper is demonstrated in a scenario built around agent-based systems. The scenario is built to be as simple as possible, yet complex enough to include examples of emergent behavior. In this scenario, we work with a cellular environment that is represented as a large matrix of adjacent cells. The cells can be occupied by three different entities: (1) *agent*, (2) *load*, and (3) *destination*. An agent is commissioned to pick up the nearest load and deliver it to the nearest destination. The agent has always all the necessary information regarding the positions of the loads and the positions of the destinations. There are four agents deployed in the environment, as shown in Figure 3.

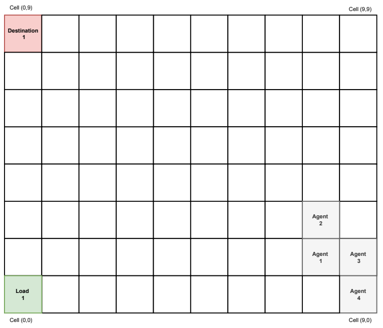


Fig. 3. Agent-Based scenario (Initial state)

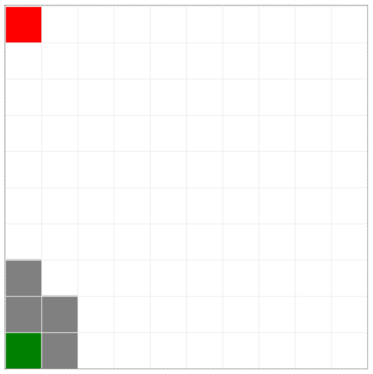


Fig. 4. Simulation of the studied Agent-Based scenario

Even though all the agents are working properly, emergent behavior manifests itself eventually in the form of a deadlock. This can be seen in the scenario simulation results depicted in Figure 4. The deadlock is considered to be an emergent behavior because the agent is stuck in the *Idle* state, a behavior that is not specified in the requirements model or in the design model. This emergent behavior indicates the incompleteness of the development models. The conceptual framework of this paper uses the emergent behavior to extract the missing specifications and use these to complete the specification of the development models. The following section discusses each step in the conceptual framework and depicts small excerpts to show the main result of each step.

B. Development Artifacts

As show in Figure 2, the following artifacts are considered in the conceptual framework: (1) the requirements model, (2) the design model, (3) the implementation model, and (4) the mined LTL specification.

1) Requirements Model: The requirements model consists of requirements for all three entities that exist in the environment, as well as overall requirements with respect to the environment. Here are some examples of each type of requirements:

- **ENV R1:** A cell does not contain two loads or more or two agents or more at the same time.
- **ENV R2:** The four agents are positioned in the following cells: (9,0)(9,1)(8,2)(8,1).
- **AG R1:** An agent can occupy one cell at a given time.
- **AG R2:** An agent can move in the four directions: *Top*, *Bottom*, *Right*, *Left*.

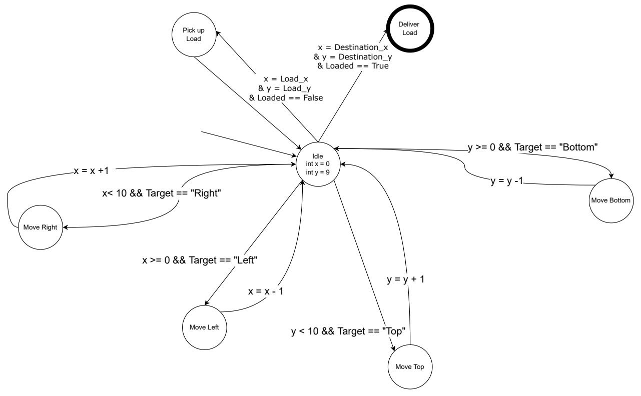

Pick up Load
Deliver Load
x = Destination_x & y = Destination_y & Loaded == True
x = Load_x & y = Load_y & Loaded == False
Idle int x = 0 int y = 9
y >= 0 && Target == "Bottom"
y = y -1
Move Bottom
x = x +1
x< 10 && Target == "Right"
Move Right
x >= 0 && Target == "Left"
x = x - 1
Move Left
y = y + 1
y < 10 && Target == "Top"
Move Top

Fig. 5. Design Model as a Finite State Machine for Agent 1

- **LD R1:** A load can be picked up only by one agent at a given time.

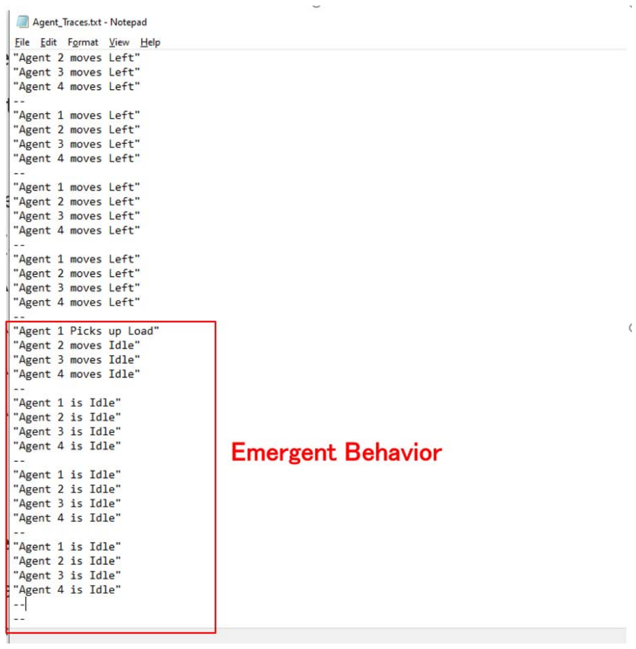
These requirements are manually transformed into LTL specification. This formalization of the requirements is supposed to help reason about the emergent behavior later and enable the specification completion in a systematic way. Examples of the LTL specification are ***atomic propositions:***

- a_1^i: Agent i moves left.
- a_5^i: Agent i Picks-up a load.
- l_2^i: Load i is delivered.

and ***formulas:***

- Φ_1: $\mathbf{F}(a_5^i)$ = Eventually agent i picks up a load.
- Φ_5: $(a_7^i) \rightarrow \mathbf{X}(l_1^i \;\&\; a_5^i)$ = if Agent i is in loading position, then the load is picked up by the agent in the next step.
- Φ_6: $(a_5^i) \rightarrow \mathbf{F}(a_5^i)$ = if Agent i picks up a load, it will eventually deliver it.

2) Design Model: For the design model, finite-state machines are considered to be the best candidate formalism for modeling the agents of the system. In Figure 5, a refinement of the requirements model is achieved through the definition of the set of states and the transitions between these states. Notice that the *Idle* state is defined even though it is not part of our requirements model. This is necessary to model the movements actions in the different four directions at each given time. At first, each agent is assumed to have full knowledge of the environment, including the positions of the loads and destinations in the environment. At the beginning, each agent calculates at each step the shortest distance to the nearest load, until an agent picks up a load. Based on the calculation results, the variable *target* takes a value of $\{$*Left, Right, Top, Bottom*$\}$, and the agent moves accordingly and updates the local (x,y) coordinates. If an agent finds itself in a loading position - which is calculated straightforward - a Boolean flag *Loaded* is set to *true*, and the agent picks up the load. Similarly, once the flag *Loaded* is set to *true*, the agent calculates the shortest path to the nearest destination and moves toward it, updating the local coordinates along the path. Eventually, the agent is placed in the destination position with the load right beneath it and the load is considered to be delivered. Each of the four agents is verified separately using the model-checking tool NuSMV [20]. Each agent fulfills all the requirements specified in the requirement model. This is also verified via the simulation where it can be see that each of the agents tested separately would successfully pick up the load and deliver it.



```
"Agent 2 moves Left"
"Agent 3 moves Left"
"Agent 4 moves Left"
--
"Agent 1 moves Left"
"Agent 2 moves Left"
"Agent 3 moves Left"
"Agent 4 moves Left"
--
"Agent 1 moves Left"
"Agent 2 moves Left"
"Agent 3 moves Left"
"Agent 4 moves Left"
--
"Agent 1 moves Left"
"Agent 2 moves Left"
"Agent 3 moves Left"
"Agent 4 moves Left"
--
"Agent 1 Picks up Load"
"Agent 2 moves Idle"
"Agent 3 moves Idle"
"Agent 4 moves Idle"
--
"Agent 1 is Idle"
"Agent 2 is Idle"
"Agent 3 is Idle"
"Agent 4 is Idle"
--
"Agent 1 is Idle"
"Agent 2 is Idle"
"Agent 3 is Idle"
"Agent 4 is Idle"
--
"Agent 1 is Idle"
"Agent 2 is Idle"
"Agent 3 is Idle"
"Agent 4 is Idle"
--
--
```

Fig. 6. Observed behavior Traces including the emergent behavior traces

3) Implementation Model: The next refinement step consists of transforming the design model, that is represented as a finite state machine, into an implementation model. The implementation is in the form of Python code. The implementation model is developed with the help of the MESA framework [21], which is a Python-based framework for the simulation of multi-agent systems. Figure 4 depicts the deadlock that eventually happens after eight computation steps. Agent 1 arrives first to the load and picks it up. However, it cannot move in any other direction, because the rest of the agents are occupying the neighbor cells. On the other hand, the other agents are seeking to pickup the load even though it has already been picked up by agent 1. This happens because the specification of the requirements model and of the design model does not require an agent to clear the way once a load is picked up. Figure 6 shows an excerpt of the behavior traces that are observed during system operation.

4) Mined LTL Specification: As explained in Section III-B, specification mining can be applied to help us reason better about the observed behavior. In the scenario of this paper, we formalize the observed emergent behavior that is highlighted in Figure 6. By applying TEXADA [13] on the observed behavior traces we can obtain multiple LTL specifications including the following ***mined LTL specification:***

- *GF"Agent 1 is Idle".*
- *F"Agent 1 picks up Load 1".*
- *F"Agent 2 moves Load 1".*

5) Difference Automaton: Using the mined LTL specification for specification completion at system design level consists of building a difference automaton between the FSM of the design model and the FSM corresponding to the mined LTL specification. The difference automaton is provided as input to the specification completion tool in order to complete the specification that causes the emergent behavior of the system. In this scenario, the emergent behavior is modeled as a simple FSM that consists only of the state *Idle* and a transition to the same state with a silent event. The emergent behavior consists of permanently executing this loop and remaining in the *Idle* state. Notice that this loop is not specified in the design model shown in Figure 5.

6) Specification Completion: The tool responsible for the specification completion checks first if the observed behavior is an emergent or not. This takes place by comparing the LTL specification of the developed requirements model against the LTL specification mined from the observed behavior traces. By definition, the emergent behavior is the behavior that is observed but not specified during development. Hence, the tool recognizes the non-specified specifications and uses them either to: (1) explicitly include the new (mined) specification in the LTL requirement model, which means that we accept the emergent behavior to be part of the system, or (2) develop new specification and add it to the requirement model in order to explicitly exclude the emergent behavior from emerging in the subsequent DevOps loops. In this way, an emergent system is a system that can automatically develop new specifications and include them into its developed models, using as input the emergent behavior observed at run-time. In the presented multi-agent example, there are different ways in which the deadlock (emergent behavior) can be avoided. For example, the specification of the development models can be adjusted to require an agent to move in a random direction in case the shortest path to the load is not available. From the point of view of the requirements model, there are multiple specifications that can be included in it in order to avoid the appearance of emergent behavior in the following DevOps iterations. The remaining challenge is how to automatically find such a specification for the purpose of specification completion using as inputs the developed and the mined specifications of the system.

VI. Conclusion

The DevOps paradigm has been proven to be practical in different multidisciplinary software development projects. This paper provides a conceptual framework that benefits from this paradigm as a way to harness the emergent behavior in a given system. The continuous development and operation process provides an opportunity to both developers and operators to make the most out of an emergent system behavior. In this regard, the novel contribution of this paper is a conceptual framework for specification completion that can be used continuously during the feedback phase to close the DevOps loop more efficiently. Furthermore, the adoption of a model-driven approach throughout the whole system development process and the formalization of the operations phases following this approach should provide a possibility for the systematic and automated completion of the missing system specifications that were not defined during development. Eventually, the emergent system is developed in different parts based on the emergent behavior and the emergent specification is always verifiable at any phase of the development process loop.

In the future, we plan to extend our approach so that the emergent behavior is used to automatically generate emergent specification, that is then used to complete the already existent specification of the system development models. This approach is based on the specified overall goal, the specification of the parts of the system and the mined specification based on system observations during its operation. At the design level, we plan on using automata mining based on model inference techniques in order to complete the specification at a design level. Furthermore, we plan of integrating suitable testing frameworks like the one proposed in Section IV-B3 to harness the emergent behavior in the different testing activities and various test phases such as the unit testing and integration testing.

VII. Acknowledgement

The results of this contribution are based on the work of the project “DevOpt: DevOps for Self-Optimizing Emergent Systems”. DevOpt is funded by the Federal Ministry of Education and Research (BMBF) of Germany in the funding programme of “IKT 2020 – Forschung für Innovationen”.

References

[1] M. Brambilla, J. Cabot, and M. Wimmer, “Model-driven software engineering in practice,” *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[2] E. M. Clarke, T. A. Henzinger, and H. Veith, “Introduction to model checking,” in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Cham: Springer International Publishing, 2018, pp. 1–26.

[3] M. Fowler, J. Highsmith *et al.*, “The agile manifesto,” *Software development*, vol. 9, no. 8, pp. 28–35, 2001.

[4] J. Allspaw and P. Hammond, “10+ deploys per day: Dev and ops cooperation at flickr,” in *Velocity: web performance and operations conference*, 2009.

[5] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, “A survey of devops concepts and challenges,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–35, 2019.

[6] M. T. Ailane, C. Knieke, and A. Rausch, “How to extend the abstraction refinement model for systems with emergent behavior ?” in *Annual Conf. on Computational Science and Computational Intelligence (CSCI22)*, 2022. [Accepted].

[7] B. Combemale and M. Wimmer, “Towards a model-based devops for cyber-physical systems,” in *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Springer, 2019, pp. 84–94.

[8] J. Hugues and J. Yankel, “From model-based systems and software engineering to moddevops,” Carnegie Mellon University's Software Engineering Institute Blog, Nov. 22, 2021. [Online]. [Online]. Available: http://insights.sei.cmu.edu/blog/from-model-based-systems-and-software-engineering-to-moddevops/

[9] J. Hugues, J. Yankel, J. Hudak, and A. Hristozov, “Twinops: Digital twins meets devops,” CARNEGIE-MELLON UNIV PITTSBURGH PA, Tech. Rep., 2022.

[10] S. A. Seshia, “Explorations in cyber-physical systems education,” *Commun. ACM*, vol. 65, no. 5, p. 60–69, apr 2022. [Online]. Available: https://doi.org/10.1145/3490442

[11] B. J. Keller and R. E. Nance, “Abstraction refinement: A model of software evolution,” *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 3, pp. 123–145, 1993. [Online]. Available: https://onlinelibrary.wiley.com/doi/epdf/10.1002/smr.4360050302

[12] M. A. Kabir, J. Han, M. A. Hossain, and S. Versteeg, “Specminer: Heuristic-based mining of service behavioral models from interaction traces,” *Future Generation Computer Systems*, vol. 117, pp. 59–71, 2021.

[13] C. Lemieux, D. Park, and I. Beschastnikh, “General ltl specification mining (t),” *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 81–92, 2015.

[14] J.-G. L. Q. F. Shengqi and Y. J. LI, “Mining program workflow from interleaved logs,” in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '10)*, 2010.

[15] A. W. Biermann and J. A. Feldman, “On the synthesis of finite-state machines from samples of their behavior,” *IEEE transactions on Computers*, vol. 100, no. 6, pp. 592–597, 1972.

[16] T.-D. B. Le and D. Lo, “Deep specification mining,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 106–117.

[17] Y. Gao, M. Wang, and B. Yu, “Dynamic specification mining based on transformer,” in *International Symposium on Theoretical Aspects of Software Engineering*. Springer, 2022, pp. 220–237.

[18] J. Shi, J. Xiong, and Y. Huang, “General past-time linear temporal logic specification mining,” *CCF Transactions on High Performance Computing*, vol. 3, no. 4, pp. 393–406, 2021.

[19] A. Aniculaesei, F. Howar, P. Denecke, and A. Rausch, “Automated generation of requirements-based test cases for an adaptive cruise control system,” in *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. IEEE, 2018, pp. 11–15.

[20] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “Nusmv: a new symbolic model checker,” *International journal on software tools for technology transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[21] D. Masad and J. Kazil, “Mesa: an agent-based modeling framework,” in *14th PYTHON in Science Conference*, vol. 2015. Citeseer, 2015, pp. 53–60.