Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Technology 00 (2016) 000–000

Procedia Technology

www.elsevier.com/locate/procedia

3rd International Conference on System-integrated Intelligence: New Challenges for Product and Production Engineering, SysInt 2016

Distributed Execution of Scenario-Based Specifications of Structurally Dynamic Cyber-Physical Systems

Joel Greenyer[a], Daniel Gritzner[a,*], Guy Katz[b], Assaf Marron[b], Nils Glade[a], Timo Gutjahr[a], Florian König[a]

[a]*Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany*
[b]*Weizmann Institute of Science, Herzl St 234, Rehovot, 7610001, Israel*

Abstract

Cyber-physical systems are found in many areas, e.g., manufacturing or smart cities. They consist of multiple components that cooperate to provide the desired functionality. This need for cooperation causes complex interactions between components, which makes developing cyber-physical systems difficult, especially systems whose component structure changes dynamically at runtime. We have created a formal, scenario-based specification method which makes it easier to develop distributed cyber-physical systems. We previously presented an approach for the distributed execution of such specifications based on naive and inefficient broadcasting. In this paper we propose a more efficient approach which uses the available network resources more economically.

Keywords: scenario-based specification; distributed execution; cyber-physical system; dynamic structure

1. Introduction

In many areas, for example, transportation, industry, and healthcare, we find cyber-physical systems (CPSs). A CPS consists of multiple cooperating software-intensive components. Together these components control complex processes and handle user interactions. Each system function is generally provided by the cooperation of multiple components and each component may be involved in the realization of multiple functions at the same time. This causes complex interactions between the components in a CPS. As customers demand increasingly rich functionality, developing these inter-component interactions becomes increasingly difficult and error-prone.

This problem is even amplified when the component structure changes at runtime: in many CPS, for example mobile robot systems, reconfigurable automation systems, or systems of coordinating vehicles, the physical or logical relationships between components or component properties change frequently during operation. This structural dynamism creates many different situations in which the components are required to interact differently, and it therefore further increases the system complexity.

[*] Corresponding author. Tel.: +49-511-762-19673.
E-mail address: daniel.gritzner@inf.uni-hannover.de

In order to help engineers cope with the complexity of developing structurally dynamic CPSs, we are developing a formal method for specifying and analyzing the behavior of a structurally dynamic system. Our method is based on (1) an object-oriented description of a dynamic component system and (2) *scenarios* that allow the engineers to specify the required and forbidden interactions of the system components. This approach fits naturally the way engineers conceive and communicate the system behavior during the early design. Our method extends the concepts of Live Sequence Charts (LSCs) [1] in a new textual language, called the *Scenario Modeling Language (SML)*.

Our method also supports the *execution* of scenarios [2,3], which helps engineers to understand and validate the behavior emerging from the interplay of the scenarios and the structural evolution of the component structure.

We are, however, pursuing *an even more radical goal*: we not only want to execute the scenario specifications offline for simulation purposes—instead we suggest to execute the specifications at runtime, so that no additional manual and error-prone programming will be necessary to implement the inter-component communication functions.

Until recently, the execution could only be realized by a centralized coordinator, which does not fit the distributed architectures of most CPSs. However, we showed how, in principle, this approach can be naively distributed by making each system component act as if it were the centralized coordinator and fully synchronizing their execution [4].

The disadvantage of this approach is clearly its communication overhead: on each event that occurs in the system, all components must be synchronized. With a growing number of components, this will rapidly congest the network. A practical approach must scale with the number of components, and must work, for example, for millions of interconnected cars; it must therefore avoid the full synchronization of all components.

In this paper we present a new practical approach for the distributed execution of scenario specifications at runtime, which uses the available resources more economically: when a scenario is executed, it only requires the synchronization of the bounded, small number of components that participate in the scenario; it is therefore scalable for systems with large numbers of components. The *key novelty* of our work is that it supports the execution of inter-component scenario-based specifications *with respect to structurally dynamic systems*. This includes concepts for maintaining, per component, a local snapshot of its relevant structural context and dynamically binding scenarios to components depending on structural properties of the component system.

This paper is structured as follows. Sect. 2 introduces an example Car-to-X system. Sect. 3 explains our scenario-based modeling and specification method. Then, in Sect. 4 we present our new distributed execution approach. Sect. 5 and 6 discuss related work and our plans for future work. The paper concludes with Sect. 7.

2. Example

As an example, we consider an advanced driver assistance system that relies on car-to-car and car-to-infrastructure communication, *Car-to-X* in short, to safely coordinate cars passing a narrow street section created by roadworks.

An intuitive way to specify the desired behavior of a system is using *scenarios*. Figure 1 illustrates two example scenarios informally—in this form, they could even be described by stakeholders with no technical background.

The scenario on the left describes that when a car approaches an obstacle (1), it must show on the car's dashboard whether or not the driver is allowed to pass the obstacle (2), and must do so before the car reaches the obstacle (3).

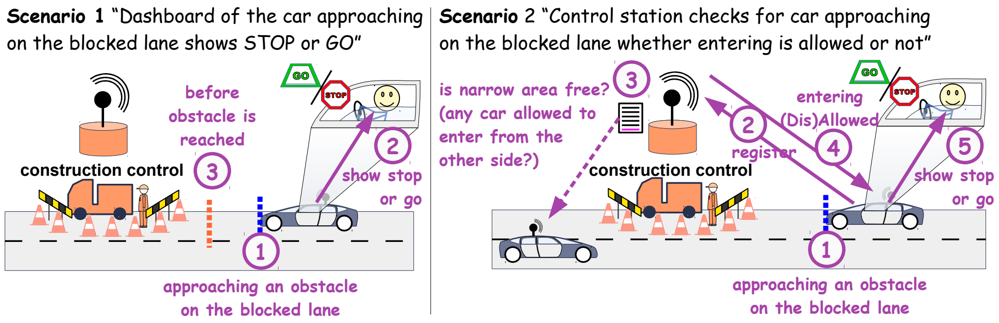

Scenario 1 "Dashboard of the car approaching on the blocked lane shows STOP or GO"
before obstacle is reached
3
construction control
GO
STOP
2
show stop or go
1
approaching an obstacle on the blocked lane
Scenario 2 "Control station checks for car approaching on the blocked lane whether entering is allowed or not"
is narrow area free? (any car allowed to enter from the other side?)
3
construction control
2
register
entering (Dis)Allowed
4
GO
STOP
5
show stop or go
1
approaching an obstacle on the blocked lane

Fig. 1. Examples of scenarios

The second scenario complements the first and describes in more detail when the car should be allowed to pass: when a car approaches an obstacle (1), it registers itself at the obstacle's control station (2). This controller tells the approaching car, based on whether the narrow passage is occupied or not (3), if it is allowed to enter the narrow passage or if it must wait (4). The result of this decision is then shown on the approaching car's dashboard (5).

A scenario-based specification may consist of many such scenarios. The scenarios can describe different situations or, as shown here, they can also overlap and describe different relevant aspects of the same situation.

3. Scenario-based Modeling

Our scenario-based design method supports the formal modeling of scenario as described above using the Scenario Modeling Language (SML). The ScenarioTools[1] tool suite supports the modeling and execution of SML scenario specifications as well as formal controller synthesis and realizability checking [5]. As a proof of concept, we already showed that these specifications can be executed on a distributed system of RaspberryPi-based robots[2] [4].

3.1. Scenario Modeling Language

SML combines the object-oriented modeling of a system's structure with a scenario-based description of the components' behavior. **Object-oriented modeling:** An SML specification refers to a class model that describes the types of components in the system, including their attributes and associations. For execution, a concrete component instance system, e.g., a concrete street system with a certain number of cars and roadworks, including attribute values and reference links, must be defined. The initial instance model defines the initial structure of the system, which may then evolve at runtime. We also call this evolving instance system the *object system*, and call a component instance an *object*. **Scenario modeling:** The behavior of the objects is modeled by a set of *scenarios*. A scenario describes a sequence of events of message exchanges between objects. A formalization of scenario 1 in Fig. 1 is shown in List. 1.

```
 1  dynamic role Environment env
 2  dynamic role Car car
 3  dynamic role Dashboard dashboard
 4
 5  specification scenario DashboardOfCarApproachingShowingStopOrGo
 6  with dynamic bindings [ bind dashboard to car.dashboard ] {
 7     message env -> car.approachingObstacle()
 8     alternative { message strict requested car -> dashboard.showGo()   }
 9     or          { message strict requested car -> dashboard.showStop() }
10     message env -> car.obstacleReached()
11  }
```
Listing 1. Example of a scenario written in the Scenario Modeling Language

A specification written in SML is a collection of *scenarios*. Each scenario has a unique name as shown in line 5. A Scenario describes how a set of objects in the object system must or must not interact, via messages, in certain situations. The objects are represented by *roles* (lines 1-3 in List. 1).

A scenario is *activated* when a message occurs in the system that corresponds to the first message of a scenario. When this happens, the sending and receiving objects of the message are *bound* to the sending and receiving roles of the scenario. For example, whenever the message approachingObstacle is sent by an object of type Environment to an object of type Car, an instance of the scenario DashboardOfCarApproachingShowingStopOrGo is created with the roles env and car being bound to the sending and the receiving object, respectively. All bindings of roles that are not the sender or receiver of the first message are defined through *binding expressions*, as shown for example in line 6 of List. 1. Here the binding expression specifies that the role dashboard shall be bound to the dashboard component that is referenced by the car that received the approachingObstacle message. This reference is part of the object system. The binding expressions are evaluated immediately after the binding of the roles of the first message and before any subsequent messages from that scenario occur.

[1] http://scenariotools.org/
[2] https://youtu.be/g0hcGSYC2Wk

Each active scenario instance has one or several pointers to messages that are expected to occur next. These messages are called *enabled messages*. After the scenario DashboardOfCarApproachingShowingStopOrGo has been activated, the messages in lines 8 and 9 are enabled. Here we have the special case of an alternative choice. Either of these messages is expected to occur next. Once one of them occurs, both are removed from the set of enabled messages and the last message of the scenario, line 10, becomes enabled. If the last message of a scenario is enabled, and a message occurs that corresponds to that message, the active scenario instance *terminates*.

If a message occurs in the system that corresponds to a message in an active scenario that is not currently enabled, a *violation* of the scenario happens, because this message was expected to occur at another time. The keyword strict determines whether such a violation is *allowed* or *forbidden*. If no enabled message is strict, the violation is allowed, but leads to immediate termination of the active scenario. If at least one enabled message is strict, the violation is forbidden. If a forbidden violation happens, we also call it a *safety violation*. Any behavior after a safety violation is undefined. With regard to List. 1 this means that the message *obstacleReached* (line 10) must not occur while the messages in lines 8 and 9 are enabled.

SML scenario messages can also be requested (lines 8 and 9) or *non-requested*. Requested messages must occur at some point in the future, i.e., they must not be enabled forever, otherwise this is a *liveness violation*. By contrast, non-requested messages need not ever occur. For example, when in the scenario in List. 1 the messages in lines 8 and 9 are enabled, one of them must occur at some point in the future, but after one of them occurred, the message in line 10 may remain enabled forever, i.e., it would be fine if the car never reached the obstacle.

A system is a *valid implementation* of a specification it if never causes safety or liveness violations of its scenarios.

There can be multiple active instances of different scenarios, and also of the same scenarios, with different bindings for the roles. By requesting and forbidding events, complex behavior emerges from the interplay of the scenarios. For the engineer this means that they can add scenarios to incrementally extend as well as restrict a system's behavior.

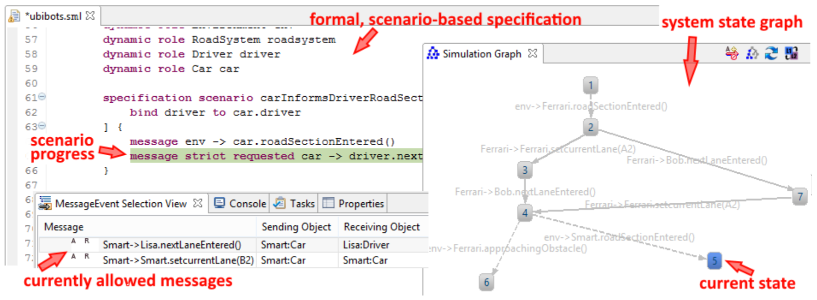
SML supports additional language constructs, such as conditions, loops, and parallel execution, which gives engineers rich possibilities to intuitively express their intentions. Also environment assumptions are supported [3].

In particular, to express structural changes, messages of the form set⟨feature⟩(⟨value⟩) can be used to change an object's attribute or link values. This way, for example, we can model that a car moves from one street section to the next. There are also further constructs for adding or removing cars from the construction control's registration list.

3.2. The Play-Out algorithm

SML specification can be *executed* via the *play-out algorithm*, which was originally introduced for LSC specifications [2]. It assumes that some objects in the object system are uncontrollable environment objects; the other objects are controllable system objects. The algorithm works as follows.

Initially, the play-out algorithm waits for a message from an environment object. This may activate one or multiple scenarios. When, as a consequence, requested messages that shall be sent by system objects are enabled, the algorithm sends one of these messages if it is not forbidden by any of the active scenarios. The algorithm repeats this execution of system messages until no requested system messages are enabled anymore. Then it waits for the next environment


*ubibots.sml
formal, scenario-based specification
system state graph
Simulation Graph
scenario progress
MessageEvent Selection View
Console
Tasks
Properties
currently allowed messages
current state
1
2
3
4
5
6
7
env->Ferrari.roadSectionEntered()
Ferrari->Ferrari.setcurrentLane(A2)
Ferrari->Bob.nextLaneEntered()
Ferrari->Bob.nextLaneEntered()
Ferrari->Ferrari.setcurrentLane(A2)
env->Smart.roadSectionEntered()
env->Ferrari.approachingObstacle()

```
    dynamic role RoadSystem roadsystem
    dynamic role Driver driver
    dynamic role Car car

    specification scenario carInformsDriverRoadSect
        bind driver to car.driver
    ] {
        message env -> car.roadSectionEntered()
        message strict requested car -> driver.next
    }
```

Message	Sending Object	Receiving Object
A R Smart->Lisa.nextLaneEntered()	Smart:Car	Lisa:Driver
A R Smart->Smart.setcurrentLane(B2)	Smart:Car	Smart:Car

Fig. 2. Environment for simulating a specification in ScenarioTools.

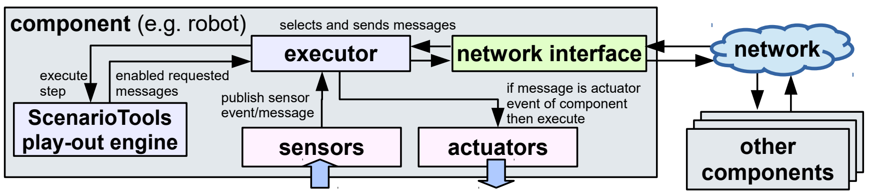

component (e.g. robot)
selects and sends messages
executor
network interface
network
execute step
enabled requested messages
publish sensor event/message
if message is actuator event of component then execute
ScenarioTools play-out engine
sensors
actuators
other components

Fig. 3. Framework for executing an SML specification.

message and repeats this process. The algorithm assumes that the system is fast enough to send any finite number of system messages between any two environment events.

Example: assume that an event approachingObstacle is received by a car of our example system. This activates a copy of the scenario DashboardOfCarApproachingShowingStopOrGo, which then requests that the car sends showGo or showStop to the car's dashboard. This event, however, will be blocked by the scenario 2 from Fig. 1 (we omit the SML listing for brevity), which will first require the car to register at the construction control, etc. Only when both scenarios reach a point where showGo or showStop shall be sent, the algorithm can send the message.

ScenarioTools supports the play-out of SML specifications and for example supports an interactive step-by-step execution within the Eclipse debug environment. Fig. 2 shows a screenshot.

To use the play-out algorithm for the actual distributed implementation of the system, it is necessary to embed the play-out algorithm in a framework that provides message communication between the components via a network. Furthermore, there must be the means to implement an interface that maps messages to and from environment objects to events of the sensors and actuators of the components. Such a framework is sketched in Fig. 3, which we implemented for our proof-of-concept demonstrator [4].

4. Distributed Execution

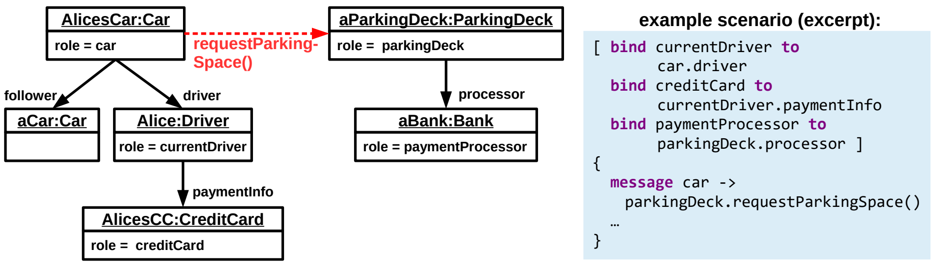
In this section, we present our algorithm for the distributed play-out of SML specifications. The core idea of the approach is that not all components synchronize on every event. Instead, the only components that synchronize are those that are bound in active scenarios where this event occurs. We call this *group-based synchronization*. The approach also has a few more elements. For example, we have to decide on what knowledge each object has about the structure of the overall system. Our pragmatic choice is that we only assume that each object knows about its own attribute values and only direct links to other objects. This avoids having to send notifications about structural changes among objects, but also demands restrictions on binding expressions and queries on structural properties, and it requires a special solution for forming the above-mentioned synchronization groups.

4.1. Group-based synchronization

Our distributed play-out with group-based synchronization requires every system object to locally execute all the active scenarios that it is involved in. Furthermore, all objects that participate in an active scenario must synchronize on each step of the scenarios in order to have a consistent perception on what must or must not happen next, i.e., which message must be sent or whether the objects should wait for the next environment event. We call these groups *synchronization groups*. At each point, every object can be in multiple synchronization groups.

When an object, due to its local play-out, decides to send a requested system message, it has to notify all other objects in the synchronization groups which are *affected* by the event. Objects in a synchronization group are *affected* by a message event if this event leads to a state change in the active scenario on which the objects synchronize, i.e., when enabled messages are processed or when the active scenario terminates.

A synchronization group could also be affected by a message from an environment object. Environment objects, however, since they are uncontrollable and do not run any local play-out logic, will not notify all objects in the affected synchronization group. Therefore, in this case, the receiving object must notify the other objects in the synchronization group about the environment message when it receives it. An object that is the sender of a system message or the



```
[ bind currentDriver to
       car.driver
  bind creditCard to
       currentDriver.paymentInfo
  bind paymentProcessor to
       parkingDeck.processor ]
{
  message car ->
    parkingDeck.requestParkingSpace()
  …
}
```

Fig. 4. Example of determining the group of objects associated to a newly created scenario instance.

receiver of an environment message is called the *master* of the message. For this approach to work, we require two restrictions: First, in all scenarios there must be no messages between only environment objects. Second, the scenarios must be specified in a way that for all enabled messages in any state of an active scenario, the master is always the same object. The latter restriction ensures that no conflicting choices on how to progress an active scenario are made.

4.2. Resolving role bindings of distributed objects

The most involved part of group-based synchronization is creating a new active scenario and making sure that all involved objects are informed about the its creation, i.e, performing a distributed resolution of role bindings.

To achieve this, the objects involved in the message activating a new active scenario start by creating this scenario instance locally. For that purpose, every object, upon receiving a message, has to not only check whether this message progresses any already active scenarios, but also has to determine if any new active scenario instances have to be created according to the specification. Every time an object creates a local active scenario instance, it has to start resolving the role bindings. Whenever a role binding is resolved by an object locally, this object notifies the bound object that it plays a role in this active scenario. Then these objects create local copies of this active scenario as well. Because objects, due to their limited knowledge about their structural context, may not be able to resolve all role bindings for an active scenario locally, this may require a distributed cascade of role binding resolutions.

An example of this process is shown in Fig. 4. Suppose the message requestParkingSpace gets sent by AlicesCar to aParkingDeck. Then both of these objects will create a local instance of the scenario that requestParkingSpace activates. Also, both objects already know their respective roles in this active scenario and can use the role binding expressions, as shown at the top of the excerpt in Fig. 4, to determine which objects they need to notify about the active scenario creation. In particular, this causes AlicesCar to notify Alice to create a local instance of the scenario with car bound to AlicesCar, parkingDeck bound to aParkingDeck, and currentDriver bound to Alice. Furthermore, aParkingDeck will send the same kind of notification to aBank and Alice will notify AlicesCC, creating a cascade of notification along the object system's structure. aCar will never be notified, despite being part of the object system, as it is not bound to any role. Consequently, aCar will not belong to the synchronization group.

During the play-out of an active scenario, message masters must know how to reach all other objects in the synchronization group. Also, when a structural change occurs, i.e., a message updates a link value of an object, this object must know how to reach the newly referenced object in order to be able to resolve role bindings correctly in the future. In SML, the parameter of such a message is the role name of the object that is being referenced after the message has been sent. For this purpose, the objects within a synchronization group need to know the mapping from role names to each involved object's network address.

Every object, which gets bound to a role in an active scenario, that is neither the sender nor receiver of the activating first message reports its role name and address back to the master of the first message. Since all roles are bound immediately after a scenario instance is activated and before any subsequent messages of that scenario are sent, eventually every participating object will have reported back to the master of the first message. Upon receiving all role bindings and addresses, this master can create a table that maps role names to addresses and transmit this table to all other objects in this group.

4.3. Properties and issues of the new approach

This new approach scales to larger systems with more components as messages are no longer broadcast globally. Consider a car convoy example, in which the lead car makes decisions for the convoy's route and communicates this to its immediate follower, which then forwards this decision and so on. In a convoy of n cars, $(n - 1)$ messages have to be sent. In our old approach, this would have caused $(n-1) \cdot (n-1)$ messages to be sent due to broadcasting. In our new approach, when ignoring the scenario instance creation overhead which only adds a constant factor, only $(n - 1)$ messages have to be sent. So we achieved an improvement from quadratic to linear scaling in this example.

Additionally, our new approach neither has a single point of failure in the form of a message broker, nor does it require the underlying network to support broadcasting.

However, there are still some issues that need to be addressed in future research. For example, scenario instance creation has a significant overhead, and this may make our old approach more efficient bandwidth-wise for small systems. Another issue is race conditions. For example, a message sent by one object may activate a scenario that forbids another message, which a different object wants to sent concurrently. This problem can be avoided by adjusting the specification.

5. Related Work

There exist other approaches for executing global behavior models in a distributed system, for example in the business process and SOA domain [6–8]. There, however, the behavior model is usually a business process model described in BPEL or activity diagrams, and not a specification of inter-component scenarios, which are better suited for the flexible specification of the reactive behavior of distributed embedded systems.

There has also been work on the specification and analysis of structurally dynamic (or reconfigurable) systems, based on graph grammars [9–11], but these do not include a joint specification of structural dynamics and scenario-based message-based interaction. In fact, we are working on this integration in ScenarioTools [12].

Abdallah et al. describe the distributed execution of Message Sequence Charts (MSCs) [13]. They, however, do not consider structural dynamism, and the composition mechanism for MSCs is not as flexible as that of LSCs or SML.

Additionally, there was work on synthesizing distributed controllers from scenario-based specifications [14–16]. However, these approaches do not consider structurally dynamic systems.

Closely related to our work is Behavioral Programming [17], which follows the same scenario-based paradigm and shares many traits with SML. The main difference is that Behavioral Programming aims to enhance existing programming languages, such as C++. This makes prototyping a system more difficult, but improves performance. There is work in the area of distributing behavioral programs [18] facing the same challenges we do.

6. Outlook and Research Roadmap

In future work we plan to move towards distributed execution algorithms that further reduce the synchronization overhead. The challenges we face are automatically determining a minimal set of necessary synchronization messages, and automatically preventing race conditions by careful message selection.

We are also working on applying the techniques proposed herein to *Behavioral Programming* (*BP*) and its distributed variants [18]. One challenge is to include the notion of a distributed object system in BP designs; a demo on initial work is available online[3]. We are currently working on a tool that can automatically transform a centralized behavioral program into an efficient distributed program with equivalent functionality.

While this paper focuses on the efficiency of distributed execution, we also plan to address other aspects of distributed systems, such as dependability or security, arising in the context of scenario-based specifications. Regarding dependability, we want to include features for the detection of and the recovery from faulty behavior. In the area of security, we are discussing new analysis techniques to help engineers with identifying possible attack strategies exploitable by malicious components, e.g., a car running modified software to gain an unfair advantage over other cars or to cause otherwise unintended overall system behavior.

[3] `http://www.b-prog.org/videos/BPDistributed.mp4`

7. Conclusion

In this paper we presented an improvement to our method for the distributed execution of SML specifications [4]. It supports the modeling and execution of scenario-based inter-component interaction behavior of structurally dynamic systems. The improvements allow our approach to scale with a growing number of components in the system.

With this execution technique, no manual programming of the inter-component communication behavior will be necessary, but specifications, where the scenarios closely resemble stakeholder requirements, can be executed directly. This greatly contributes to reducing errors in the development of complex cyber-physical systems.

We also outlined our future research efforts in continuing to improve our work.

Acknowledgements: This work is funded by grant no. 1258 of the German-Israeli Foundation for Scientific Research and Development (GIF).

References

[1] Damm, W., Harel, D.. LSCs: Breathing life into message sequence charts. In: Formal Methods in System Design; vol. 19. Kluwer Academic Publishers; 2001, p. 45–80.

[2] Harel, D., Marelly, R.. Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. Software and System Modeling 2002;2.

[3] Brenner, C., Greenyer, J., Panzica La Manna, V.. The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. In: Proc.12th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013); vol. 58. EASST; 2013,.

[4] Greenyer, J., Gritzner, D., Gutjahr, T., Duente, T., Dulle, S., Deppe, F., et al. Scenarios@run.time – distributed execution of specifications on IoT-connected robots. In: Proceedings of the 10th International Workshop on Models@Run.Time (MRT 2015), MODELS 2015. 2015,.

[5] Greenyer, J., Brenner, C., Cordy, M., Heymans, P., Gressi, E.. Incrementally synthesizing controllers from scenario-based product line specifications. In: Proc. 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2013. 2013,.

[6] Nanda, M.G., Chandra, S., Sarkar, V.. Decentralizing execution of composite web services. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA '04; New York, NY, USA: ACM. ISBN 1-58113-831-8; 2004, p. 170–187.

[7] Kunze, C.P., Zaplata, S., Lamersdorf, W.. Distributed Applications and Interoperable Systems: 6th IFIP WG 6.1 International Conference, DAIS 2006, Bologna, Italy, June 14-16, 2006. Proceedings; chap. Mobile Process Description and Execution. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-35127-6; 2006, p. 32–47.

[8] Spiess, P., Karnouskos, S., Souza, L., Savio, D., Guinard, D., Trifa, V., et al. Reliable execution of business processes on dynamic networks of service-enabled devices. In: Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on. 2009, p. 533–538.

[9] Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.. Symbolic invariant verification for systems with dynamic structural adaptation. In: Proceedings of the 28th International Conference on Software Engineering. ICSE '06; New York, NY, USA: ACM. ISBN 1-59593-375-1; 2006, p. 72–81.

[10] Klein, F., Giese, H.. Fundamental Approaches to Software Engineering: 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007. Proceedings; chap. Joint Structural and Temporal Property Specification Using Timed Story Scenario Diagrams. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-71289-3; 2007, p. 185–199.

[11] Bruni, R., Bucchiarone, A., Gnesi, S., Melgratti, H.. Modelling dynamic software architectures using typed graph grammars. Electronic Notes in Theoretical Computer Science 2008;213(1):39 – 53. Proceedings of the Third Workshop on Graph Transformation for Concurrency and Verification (GT-VC 2007).

[12] Winetzhammer, S., Greenyer, J., Tichy, M.. Integrating graph transformations and modal sequence diagrams for specifying structurally dynamic reactive systems. In: Amyot, D., Fonseca i Casas, P., Mussbacher, G., editors. System Analysis and Modeling: Models and Reusability; vol. 8769 of *Lecture Notes in Computer Science*. Springer International Publishing. ISBN 978-3-319-11742-3; 2014, p. 126–141.

[13] Abdallah, R., Hélouët, L., Jard, C.. Distributed implementation of message sequence charts. Software & System Modeling 2013;14(2):1029–1048.

[14] Harel, D., Kugler, H., Pnueli, A.. Synthesis revisited: Generating statechart models from scenario-based requirements. In: Formal Methods in Software and Systems Modeling; vol. 3393. Springer; 2005, p. 309–324.

[15] Bontemps, Y., Heymans, P.. From Live Sequence Charts to State Machines and Back: A Guided Tour. IEEE Transactions on Software Engineering 2005;31(12):999–1014.

[16] Brenner, C., Greenyer, J., Schäfer, W.. On-the-fly synthesis of scarcely synchronizing distributed controllers from scenario-based specifications. In: Egyed, A., Schaefer, I., editors. Fundamental Approaches to Software Engineering (FASE 2015); vol. 9033 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-662-46674-2; 2015, p. 51–65.

[17] Harel, D., Marron, A., Weiss, G.. Behavioral programming. Commun ACM 2012;55(7):90–100.

[18] Harel, D., Kantor, A., Katz, G., Marron, A., Weiss, G., Wiener, G.. Towards behavioral programming in distributed architectures. Science of Computer Programming 2015;98(2):233–267.