Efficient Distributed Execution of Multi-component Scenario-based Models

Shlomi Steinberg[1], Joel Greenyer[2], Daniel Gritzner[2], David Harel[1], Guy Katz[3], and Assaf Marron[1]

[1] The Weizmann Institute of Science, Rehovot, Israel
[2] Leibniz Universität Hannover, Hannover, Germany
[3] Stanford University, Stanford, USA

Abstract. In scenario-based programming (SBP), the semantics, which enables direct execution of these intuitive specifications, calls, among others, for synchronizing concurrent scenarios prior to every event-selection decision. Doing so even when the running scenarios are distributed across multiple physical system components, may degrade system performance or robustness. In this paper we describe a technique for automated distribution of an otherwise-centralized specification, such that much of the synchronization requirement may be relaxed. The technique calls for replicating the entire scenario-based executable specification in each of the components, locally transforming it in a component-specific manner, and reducing the synchronization requirements to very specific and well-defined points during execution. Our evaluation of the technique shows promising results. Given that relaxed synchronization can lead to what appears as different runs in different components we discuss various criteria for what would constitute acceptable differences, or divergence, in the parallel, distributed runs of almost-identical copies of a single specification.

This paper incorporates and substantially extends the material of the paper published in MODLESWARD'17 *Distributing Scenario-Based Models: A Replicate-and-Project Approach* by the same authors[37].

Keywords: Software Engineering, Scenario-Based Modeling, Concurrency, Distributed Systems

1 Introduction

With modern reactive systems becoming both pervasive and highly complex, modeling them is becoming increasingly difficult. Modelers are forced to spend ever-larger amounts of time and effort in order to reconcile two goals: (1) accurately describe complex real-world systems and phenomena; and (2) do so using models that are simple, comprehensible and intuitive to humans. These two goals are often conflicting: it is difficult to describe the properties of such systems accurately, while at the same time avoiding clutter, which makes it harder for humans to comprehend the resulting models.

Over the recent two decades, an approach termed *Scenario-Based Modeling* [6] has emerged as an attempt to tackle these difficulties. The idea at its core is to model

systems in a way that is more intuitive and understandable to humans — by defining *scenarios* that describe desirable or undesirable system behavior — and then to automatically combine these scenarios in a way that produces a cohesive, global model. Appropriate scenario-based approaches and tools have executable semantics, thus helping to streamline the deployment of scenario-based models in the real world.

A scenario-based approach has been claimed to be more intuitive for humans to understand (see, e.g., [11]). It allows the modeler to specify different but possibly interrelated behavioral aspects as separate scenarios, reducing the inherent complexities of the modeling process. However, by default and as explained later, a scenario-based execution requires that all scenarios synchronize at every step, for the purpose of joint event selection. When executing scenario-based specifications in a distributed architecture, inter-scenario synchronization induces inter-component synchronization, which may be undesirable in real-world systems, where communication is often costly, slow, or unreliable. This difficulty constitutes a serious barrier when considering the use of scenario-based modeling in a real-world distributed setting.

We seek to address this problem by proposing an automated technique for the transformation of classical, highly synchronous scenario-based models into equivalent models with a greatly reduced level of synchronization. The basis of our approach is a rather straightforward *replicate-and-project* (R&P) technique but with some subtle facets: we *replicate* the full set of scenarios in all the distributed components, but *project* them in a component-specific fashion, so that each component is made responsible only for the actions that fall within its the local scope. Other, external actions, are assumed to be performed by projections running on other components.

The scenarios then progress asynchronously, each selecting and triggering events almost completely at its own pace. In order to make the replicated-and-projected scenarios behave the same as their non-distributed version, the distributed components broadcast the local actions they perform to all other components. At times a situation arises that forces some of the distributed components to mutually agree on the next action to perform. This might happen either due to an exclusive choice among multiple enabled actions (i.e., events), or due to communication latency that might result in different orders of broadcast actions as observed by different components. In these cases, the affected scenarios indeed synchronize and reach a joint decision. An important part of the work in this paper is dedicated to classifying these cases, understanding when they arise, automatically detecting their occurrence in a program, and proposing practical approaches for resolving them.

This process is handled automatically by our distribution algorithm and infrastructure, and, as we discuss and demonstrate later, it aims to generate a distributed model that has as few synchronization points as possible.

The motivation behind the approach is to retain the modeler's ability to use classical scenario-based modeling, with its associated advantages, but to be able to then transform the model into a version that is more amenable to distribution and deployment in the real world. We prove that, under certain restrictions, our proposed transformation preserves the behavior of the original model. This gives rise to a methodology for developing distributed scenario-based models, where one models a distributed system as

if it were centralized, and the model is then automatically adjusted to more accurately simulate (or even run in) its final setting.

Automatic distribution of general models (i.e., not just scenario-based ones) or synthesizing distributed models from specifications have been long-standing goals of the software modeling and engineering community. Specifically, distributed synthesis is known to be undecidable in general [36]. We contribute to this effort by studying the problem in the context of scenario-based modeling, and leveraging some of the paradigm's properties of naturalness and relative simplicity. However, difficulties nevertheless arise. We classify and describe them, and explain how they can still be addressed. Our experimental results indicate that the technique holds much potential for becoming practical.

The rest of the paper is organized as follows. In Section 2 we provide a brief introduction to the scenario-based approach. In Section 3 we present an example of a distributed execution of a scenario-based specification for a light show to be performed by light-equipped drones. This general description is used in the rationale and explanation of various technical details throughout the paper. and a variant of the example is used in the technical evaluation. In Section 4 we discuss variations, which may or may not be allowed when transforming a fully synchronized execution into one where some synchronization requirements are relaxed and certain actions may occur in a different order. In Section 5 we describe the replicate-and-project technique for automatically generating an executable distributed scenario-based model from a non-distributed one. In this section we also prove the correctness of this transformation according to the criteria set in Section 4. Section 6 describes how the approach can be applied when different components in the model operate on different time scales. An example implementation and its evaluation appear in Section 7. In Section 8, we discuss related work that has been carried out on automatic distribution, both in the general setting and in the context of scenario-based modeling, Section 9 contains a discussion of our ongoing and planned future work. We conclude in Section 10.

2 Background: Scenario-Based Specifications

Scenario-based specifications were introduced with the *Live Sequence Charts* (*LSC*) formalism [6, 25]. The approach, aimed at developing executable models of reactive systems, shifts the focus from describing individual objects of the system into describing individual behaviors thereof. The basic building block in this approach is the *scenario*: an artifact that describes a single behavior of the system, possibly involving multiple different components thereof. Scenarios can describe desirable behaviors of the system or undesirable ones, and their combinations. A set of user-defined scenarios is then interwoven into one cohesive, potentially complex, system behavior.

Several facets of scenario-based modeling have been discussed and handled in different ways: scenarios can be represented graphically, as in the original LSC approach, or textually, embedded within conventional programming languages [27, 13]; scenario-based models can be executed by naïve *play-out* [26], by smart play-out with lookahead [23] or via controller synthesis (see, e.g., [29, 13]). The modeling process can be augmented by a variety of automated verification, synthesis and repair tools [21, 16].

Research has shown that the basic principles at the core of the approach, shared by all flavors, are *naturalness* and *incrementality* — in the sense that scenario-based modeling is easy to learn and understand, and that it facilitates the incremental development of complex models [11, 1]. These properties stem from the fact that modeling is done in a way similar to the way humans explain complex phenomena to each other, detailing the various steps and behaviors one at a time.

For the remainder of the paper, we focus on a particularly simple variant of scenario-based modeling, called *behavioral programming* (*BP*) [27]. Despite its simplicity, BP has been successfully used in developing medium scale projects [18, 20], and is also known to be particularly amenable to automatic analysis tools [22]. These properties render BP a good candidate for demonstrating our approach. The rest of this section is dedicated to demonstrating and formally defining BP.

In BP, a model is a set of scenarios, termed also *behavior threads*, or *b-threads*, and an execution is a sequence of points, in which all the scenarios synchronize. At every behavioral-synchronization point (abbreviated *bSync*) each scenario pauses and declares events that it *requests* and events that it *blocks*. Intuitively, these two sets encode desirable system behaviors (requested events) and undesirable ones (blocked events). Scenarios can also declare events that they passively *wait-for* — stating that they wish to be notified if and when these events occur. The scenarios do not communicate their event declarations directly to each other; rather, all event declarations are collected by an execution infrastructure common to all b-threads, termed the *event selection mechanism* (*ESM*), after its main function. Then, at every synchronization point during execution, the ESM selects (*triggers*) an event that is requested by some scenario and not blocked by any scenario. The ESM notifies every scenario that requested or is waiting for the triggered event about this selection. The b-threads can then update their internal states, and proceed to their next synchronization point. When all affected b-threads synchronize again (with each other and with the b-threads that were not affected) the ESM repeats the event selection process. In BP, this notification of all affected b-threads is the essence of event triggering. Any additional action that the designer wishes to associate with an event (e.g., opening a water tap, turning car's steering wheel, or flashing a light) is to be carried out by the individual b-threads, using arbitrary method calls, as they transition from one synchronization point to another (by contrast, in the LSC language, the triggering of an event also drives the invocation of a corresponding method provided by the application). Fig. 1 (borrowed from [20]) demonstrates a simple behavioral model.

Formally, BP's semantics are defined as follows. A scenario, also referred to as a *behavior thread* (abbreviated *b-thread*), is defined as a tuple

$$BT = \langle Q, q_0, \delta, R, B \rangle$$

and with respect to a global set of events Σ. The components of the tuple are: a set of states Q representing synchronization points; an initial state $q_0 \in Q$; a deterministic transition function $\delta : Q \times \Sigma \to Q$ that specifies how the thread changes states in response to the triggering of events; and, two labeling functions, $R : Q \to \mathcal{P}(\Sigma)$ and $B : Q \to \mathcal{P}(\Sigma)$, which specify the events that the thread requests (R) and blocks (B) in a given synchronization point.

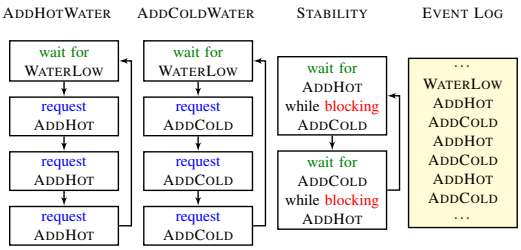

Fig. 1. Incrementally modeling a controller for the water level in a tub. The tub has hot and cold water sources, and either may be turned on in order to increase/reduce the water temperature. Each scenario is given as a transition system, where the nodes represent synchronization points. The scenario AddHotWater repeatedly waits for WaterLow events and requests three times the event AddHot. Scenario AddColdWater performs a similar action with the event AddCold, capturing a separate requirement, which was introduced when adding three water quantities for every sensor reading proved to be insufficient. When a model with scenarios AddHotWater and AddColdWater is executed, the three AddHot events and three AddCold events may be triggered in any order. When a new requirement is introduced, to the effect that the water temperature be kept stable, the scenario Stability is added, enforcing the interleaving of AddHot and AddCold events by using event blocking. The execution trace of the resulting model is depicted in the event log.

A *behavioral model* M is defined as a collection of b-threads

$$M = \{BT^1, \ldots, BT^n\},$$

all of them with respect to the same event set Σ. Denoting the individual b-threads as

$$BT^i = \langle Q^i, q_0^i, \delta^i, R^i, B^i \rangle,$$

an execution of model M starts at the initial state $\langle q_0^1, \ldots, q_0^n \rangle$. Then, at every state $\langle q^1, \ldots, q^n \rangle$, the model progresses to the next state $\langle \bar{q}^1, \ldots, \bar{q}^n \rangle$ by:

1. selecting an event $e \in \Sigma$ that is enabled, i.e. requested by at least one b-thread and blocked by none:
$$e \in \left(\bigcup_{i=1}^{n} R^i(q^i) \right) \setminus \left(\bigcup_{i=1}^{n} B^i(q^i) \right)$$
2. triggering event e and advancing the individual b-threads according to their transition systems:
$$\forall i, \qquad \bar{q}^i = \delta^i(q^i, e)$$

For reactive systems, executions are usually considered to be infinite, although BP can also be used to model systems with finite executions.

The BP definitions above are generic, making it easier to reason about behavioral models. However, for practical purposes, the BP modeling principles have been integrated into a variety of high-level languages such as Java, C++, Erlang and Javascript (see the BP website at `http://www.b-prog.org/`). These frameworks allow engineers to integrate reactive scenarios into their favorite programming or modeling environments. Further, the same principles that underly BP play a significant role in several popular modeling frameworks, such as *publish-subscribe* architectures [8] and *supervisory control* [35].

We conclude the section by defining the global state (the *cut*) of a behavioral model that is being executed:

Definition 1. *Given a behavioral model* $M = \{BT^1, \ldots, BT^n\}$*, the* program cut $r \in Q^1 \times \cdots \times Q^n$ *is defined to be the current model state:* $r = \langle q^1, \ldots, q^n \rangle$ *where* q^i *is the current state of b-thread* BT^i*.*

3 A Running Example

In many complex multi-participant operations, the participants, be they mechanical entities or people, have to carry out actions in turns, one participant after the other. A typical example is the all-way-stop traffic intersection (a.k.a. four-way stop). When there are queues in each of the intersecting roads, the cars cross the intersection one at a time, in a round-robin fashion, each coming from the front of the next queue. Another example is an audience in a packed stadium 'doing the wave', where groups of people stand up briefly and then sit down, in sequential order. These behaviors are very easily described using scenario-based specifications, where the most basic behavior can be described with a single scenario showing all the relevant entities performing their required actions in turn. (Of course, there are also other kinds of scenarios; e.g., for passing a all-way-stop intersection when you are the only car, or for the starting or the ending of a stadium wave by an audience.)

The example that we will use both to illustrate our general considerations and as the subject of our detailed analysis, is a simple drone-based light show (see elaborate shows by Disney in `www.youtube.com/watch?v=gYr-PO9meHY`, and by Intel in `www.youtube.com/watch?v=teQwViKMnxw`). In our case, a set of drones form a circle and flash their respective lights in successive turns, creating the appearance of a point of light moving in a circle. More details are provided in Section 7. The basic, single-cycle example is then expanded into repeating the cycle, stopping the cycle and then restarting it with a different, arbitrarily-selected drone, and having multiple concurrent cycles where each drone is equipped with multiple lights, perhaps of different colors.

In considering this example one may also think of analogies to human behavior: replacing the programmer or designer with a show director, the drones with people, perhaps children, who play roles in the show, and the computerized scenarios or programs (as well as the underlying SBP infrastructure), with the instructions given by the director to the participants about what they should do, and when. The autonomous starting of a new cycle at arbitrarily-selected drones may also be considered as reacting to an uncontrolled environment event, e.g., when the show-director decides on their own and unpredictably, at run time, which drone will be the first in the next cycle, and then signals it to do so.

4 Distributing a Centralized SBP Execution: Success Criteria

4.1 Success Criteria

In order to assess the properties of a distributed execution of a specification that was originally written with centralized-execution semantics assumptions, we first discuss (a) formalization assumptions, namely: which physical properties of the distributed environment will be reflected in the formal solution and which will be abstracted away,

and (b) criteria for what constitutes a correct, desirable, or perhaps just acceptable, distributed execution.

In a centralized system the concept of a run is well defined and intuitive as the sequence of system states and generated events. In a distributed environment, especially one that includes replication, this very definition is no longer without question. E.g., is the distributed run the collection of local runs as executed and observed in each component? Or perhaps it is a sequence of only the triggered events, without state transitions, ordered according to the occurrence of events in the real world, e.g., according to a time order as defined by fully synchronized component-specific clocks? Or should yet another definition be applied?

Once defined, what would be the desired properties of such a distributed run? Clearly, our goal is that it be materially different from a run where all distributed components fully synchronize before every event, and that some independent local progress will be allowed in each component. So the question we are facing is this: how much should the execution of the various components be allowed to vary from each other?

In subsequent sections we offer a particular set of principles for the sought-for solution, and a distribution mechanism that satisfies them. But beforehand we first discuss a broader list of candidate principles from which the above were chosen. Some of these can be formally defined and then examined both by model-checking and by run-time monitoring. Each of the candidate principles is accompanied by two examples - one demonstrating its desirability, and one showing that acceptable distributed executions exist that do not satisfy this principle, and hence it cannot be required of all distributed executions.

Constant Composite-State Consistency. This principle assumes that all scenarios are replicated in all components (as proposed in the present paper), and requires that the all components go through exactly the same orchestrated state transitions, and hence observe the same runs (even if not exactly at the same time). The replicated run is also a legal run of a centralized or fully synchronized execution. This could be desired and applicable, for example, in an application that has only pre-programmed actuation (as in the most basic drone light show example). Clearly this could not be demanded in a case of a reactive application with distributed input sensors, where two environment events can occur in two distinct components, one after the other, but with a time difference that is smaller than the inter-component communication delay, and where the sensing component has to acknowledge the event receipt even before all other components learned about it. As a result, the runs of the two sensing components will be different — each having a state where its own sensing scenario has sensed the event and changed its state, but none of the others have done so.

Always Eventually Reaching Composite-State Consistency. Under this principle, the entire specification is replicated as before, but components' runs are allowed to diverge as long as there is at least one composite state that each component reaches infinitely often. In other words, the components may diverge, as long as sooner or later they maintain the same view of reality (not necessarily at the same time). An example of such an application can be seen in a distributed application of industrial robots performing manufacturing tasks in parallel on a large piece of sheet metal, where the order of events across robots is not critical as long as all components are occasionally

synchronized and are at the same state (e.g., when releasing the finished piece of sheet metal and moving on to the next one). This however will not satisfy what is needed for a highly orchestrated robotic collaboration (and not even for the basic drone light show).

Distinguishing *What* and *How* Scenarios. This principle views specifications as being divided into scenarios that specify the criteria for success of the system's operation, i.e., what the system should accomplish, and scenarios that specify how the system should accomplish these goals. In classical programming the *what* scenarios appear in requirements documents and test plans, and the *how* instructions constitute the application. Research in areas such as automated program synthesis attempts to automatically generate the details of the *how* from the specification of the *what*. Here we propose to specify and retain both sets of scenarios, but require that only the *what* scenarios must be complied with in the distributed run, while the *how* scenarios can be violated in the distribution process. In the light-show example, the show director wants to achieve the appearance of cycles, or perhaps even just the appearance of pretty patterns. He or she may not care if certain drones flash their lights out of order, especially if they are in close proximity, the successive inter-drone flash delays are short, and the duration of each flash is much longer than this inter-drone delay; a drone that misses its cue may also be allowed to avoid flashing altogether in a given cycle; a drone whose battery runs out may leave the show altogether; and, neighbors of a failing drone may change their behavior as well . Thus, the divergence of runs among various components may be unbounded, while the show goes on successfully.

An example of when this approach cannot be applied can be seen in the following: a show director and an engineer created an elaborate show whose specification contains many scenarios. For testing purposes the show was implemented on a single computing component with multiple physical lights. The show is elaborate and its specification gradually evolved to have many scenarios. The director has now left, and the engineer has been tasked with distributing the implementation to the separate drones. As far as the engineer is concerned, the entire specification is the *what*, everything that was done in the centralized execution should be done in the same way in the distributed version — he or she does not know which of properties were considered essential by the director, and which can be compromised.

Language Equivalence. Under this principle we do not care about run variation among the components. Instead, we only look at the sequence of events produced (triggered) by the system, in all components, as ordered in the real world, according to some global time stamp. In this case, we do not require that a particular run of the distributed environment, defined in this manner, be equivalent to a particular run of the single-component system, but assume that there is some nondeterminism in both implementations, and simply require that the two languages, each containing *all runs* of the implementation, be equal. Thus the nondeterminism implied by the underspecification that is already built into the original requirements will be exploited by the variation imposed by the non-synchronized, sometimes-delayed, distributed execution. This can indeed be viewed as a variant of the previous principle, where the existence (in the centralized execution) of synchronization points where more than one event is enabled, is taken to be an explicit specification that selecting any of them would be acceptable. (We

assume that the event selection strategy is random, and that the application was verified with all possible combinations of event selection.)

4.2 Semantic Consideration

It goes without saying that a success criteria to be added to the above is that the execution should comply with the basic BP semantics, in that, e.g., only requested events are triggered, and events that are blocked are not to be triggered. The solution that we propose in the coming sections satisfies this basic requirement with one exception: *There is no reliance on cross-component blocking of already-enabled events.* Clearly, when an event is triggered, two b-threads may change their states, where one will start requesting a particular event e_1, and another will start blocking that same e_1. The effect of such blocking is immediate. This semantics is generally preserved in the solution we will describe, e.g., when these two b-threads change states together, in response to a single event, within a single component. However, we introduce an assumption that relaxes this requirement in that it allows event blocking to not take hold in the following case: An event e_0 causes a b-thread BT_1 to change states and start requesting event e_1. An event e_2 is then triggered, and causes b-thread BT_2 to change state and start blocking e_1. If after event e_2 occurred in one component, but before this event reaches a component requesting e_1, e_1 is already triggered in that component, we do not consider it a violation of the specification or of the BP semantics. Another way to look at this relaxation of the semantics is that it assumes that the application does not rely on the ability of one component to force the blocking of already-enabled, not-previously blocked events in other components, in time, before they are triggered.

For illustration, consider the following example: a robot-driven car is approaching an intersection, and in order to avoid collisions it must communicate with other cars. However, if the communication happens just before entering the intersection, any delay or missed messages could result in an accident.

In order to avoid this kind of issue, programs designed for distribution should employ design patterns and methods that take a realistic communication delay into account. E.g., checking for other cars early, while approaching the intersection, rather than, say, relying on scenarios to block all events of cars entering the intersection following the occurrence of an event reporting that one car already entered that intersection. We feel that this is a valid assumption in designing distributed systems and does not contradict or make redundant the advantages of BP.

This assumption, formalized in Section 5, can thus be seen as a restriction on how the application should be coded, or on features that must be added to the application if not already written in this manner.

4.3 Additional Considerations

As distributed implementations introduce new risks, additional responsibilities have to be imposed both on the distribution mechanism and on the application scenarios themselves.

Robustness. There is a desire to minimize the probability of error and of failure. First, we would like the scenarios governing the behavior to be as simple as possible.

Second, 'the show must go on' even if one of the participants made a mistake or missed their cue. For the latter, specific scenarios can probably be added. In the light show example, we could add "when a drone observes that a predecessor drone has failed or is delayed, it should nevertheless continue the cycle.". **Efficiency.** Often, the joint operation should also be required to be efficient. Consider for example the case when many bricks have to be moved from point A to point B over a narrow passage. A group of robots may be arranged in a row — passing bricks from one to the next, rather than each one traveling the entire distance. The scenarios should be designed so that inter-scenario synchronization and coordination is minimized, or decreased, and both scenario progression and the physical motion of bricks occur in parallel, asynchronously. Such measures of efficiency are evaluated in the example in Section 7

5 Distribution via Replicate-and-Project

The execution of a classical BP model, as described in Section 2, is highly synchronized and centralized by nature: at every step along the execution, the ESM gathers the sets of requested and blocked events from each individual b-thread, selects an enabled event (i.e., requested by some b-thread but blocked by none), and broadcasts it back to the b-threads. While this underlies some of the benefits of BP [27], it also results in limited scalability and distributability. Excessive synchronization tends to add unnecessary complexity, impact performance, and create inter-component dependencies that reduce robustness. For example, having a scenario wait for an event that is supposed to be requested by a scenario running on a separate, failed component might result in deadlock. Furthermore, synchronization forces b-threads to execute in lockstep, which can be undesirable if they are to model phenomena that occur at different timescales.

In this section we propose a *distribution process* that, given a centralized (undistributed) behavioral model, generates a distributed one: It creates multiple *component* models — subsets of the original, centralized behavioral model — each a behavioral program, designed to be run on a separate machine. Run simultaneously, these behavioral component models (or simply, component models) mimic the behavior of the original system, but require much less synchronization. Below we elaborate on the abstract concepts and formal definitions of the proposed process.

Each of the component models produced by our distribution process is a behavioral model in its own right, intended to be responsible for a certain subset of the events of the original model, which are uniquely owned and *controlled* by it — meaning that no other component can request or block them. The behavioral component models are intended to be executed in an asynchronous manner in a distributed system, resulting in a natural, robust and simple extension of the scenario-based paradigm.

The main difficulty in this approach is to ensure that the distributed components behave in the same way as the original model although they are not synchronized at every step. In mitigating this difficulty, the crux of our distribution process is the replication of the entire set of original scenarios in each of the distributed components, granting the components the ability to follow what other components are doing, but avoiding synchronization whenever possible. First, there is no central, coordinating ESM. Every component runs a separate, local, ESM, which by default, performs local event selec-

tion without synchronizing with other components. However, at every synchronization point where multiple components have to agree on the particular event to select, the ESMs of these components do synchronize.

The communication between components is asynchronous, and they notify each other about chosen events as they progress through the scenarios. Keeping track of each scenario state is simply a matter of listening to incoming broadcasts and updating the current state. This asynchrony is a cornerstone of the process, allowing us to generate true concurrent distributed models.

The classical problem of multicasting or broadcasting a message efficiently in a distributed network is well studied (for example, the authors of [33] present an approach for minimum-energy-broadcasts in distributed networks with limited resources and unknown topology). However it is beyond the scope of this paper. For simplicity we assume that the cost of those broadcasts and bookkeeping is small. Note that even in systems with a large number of components and scenarios, a component often needs to keep track of only a small subset of the other components; for example, an autonomous car considers other cars only when they are in its immediate vicinity, and does not have to keep track of all the vehicles in the world. Still, this dynamic registering and unregistering of components is also beyond the scope of this paper and is left for future work.

In the remainder of the section we formalize these notions and the distribution process itself.

5.1 Defining Event Components

Let M denote a behavioral model over event set Σ. An *event component* E is a subset of the global event set, $E \subseteq \Sigma$. Intuitively, each subset E reflects (or is implicitly defined by) a physical component of the distributed system and its responsibility in terms of physical capabilities and/or environment interfaces, i.e., sensors and actuators, that this component has. An event $e \in E$ is said to be a *local event* of E; otherwise, if $e \notin E$ then e is *external* to E.

A collection of event components $\{E_1, \ldots, E_k\}$ is an *event separation* of Σ if $\bigcup_{i=1}^{k} E_i = \Sigma$. An event separation is *strict* if it also forms a partition of Σ:

$$\forall\, i, j, \quad 1 \leq i \neq j \leq k \implies E_i \cap E_j = \emptyset.$$

In the remainder of the paper we will only deal with strict event separations and assume that they are provided by the user. Automated ways of generating an event separation are discussed in section 8.

5.2 Creating Behavioral Component Models by Replication and Projection

Given a behavioral model $M = \{BT^1, \ldots, BT^n\}$ over event set Σ and a strict event separation $\{E_1, \ldots, E_k\}$, each event component E gives rise to a *behavioral component model* C, in the following way. C is the behavioral model $C = \{BT_E^1, \ldots, BT_E^n\}$, obtained by *projecting* each of the original b-threads onto event component E. The projection

operation, denoted as $C = project(M, E)$, transforms each of the original b-threads as follows. If $BT^i = \langle Q^i, q_0^i, \delta^i, R^i, B^i \rangle$ then

$$BT_E^i = \langle Q^i, q_0^i, \delta^i, R_E^i, B_E^i \rangle$$

is defined as follows: The state set Q^i, the initial state q_0^i and, most importantly, the transition function δ^i which specifies how events cause state transitions, are replicated by the projection process without change. The original labeling functions R^i and B^i, namely the sets of requested and blocked events in each state, are projected onto the respective R_E^i and B_E^i according to the rules:

$$R_E^i(q) = R^i(q) \cap E$$
$$B_E^i(q) = B^i(q) \cap E$$

That is, the projected b-threads are modified to request and block only events that are in E; but because δ^i is unchanged they continue to respond in the same way to the triggering of all events, including those not in E. Consequently, where an external event is requested in a b-thread, it is modified to only be waited-for.

Now, given a (strict) event separation $\{E_1, \ldots, E_k\}$, our distribution process entails projecting the model M onto each of the event components, producing a set of component models $\{C_1, \ldots, C_k\}$ such that

$$\forall i \; 1 \leq i \leq k, \qquad C_i = project(M, E_i)$$

By treating each component C_i as a separate behavioral model that performs event selection and scenario advancement (i.e., state transition) locally, the components can be run independently and in a distributed manner. This is, however, qualified by the fact that, in order to keep the execution consistent between components, at certain points two or more components need to synchronize with each other. This is discussed in detail in the next subsection.

The following useful corollary is a direct conclusion that arises from the definition of the distribution process, when applied in the context of strict event separations.

Corollary 1. *An event $e \in \Sigma$ can be selected by at most one component.*

Proof. $\{E_1, \ldots, E_k\}$ is a strict event separation, hence there is only one value of i such that $e \in E_i$. Only C_i can request e, since, by the definition, in all other components C_j, $j \neq i$, the requests for e are replaced by waiting for it. Therefore only C_i can select e. □

5.3 Distributed Execution of Replicated-and-Projected Component Models

As discussed in Section 4, despite their parallel asynchronous execution, it is our goal that component-model execution be consistent with each other and with that of the original model. Since in the specification more than one event may be requested at a given state, occasionally these distributed runs need to be synchronized. In this subsection

and the next we describe the mechanics of parallel distributed execution of component models, and the specific synchronization constraints this execution is subjected to.

The R&P approach includes using in each component a modified BP execution infrastructure. The component's ESM is different from the one described in Section 2, in that it broadcasts to other components its local independent decisions, it processes similar messages received from other components, and, when required, it synchronizes with other components to make a joint decision.

Specifically, the following rules govern each component's ESM and the distributed execution.

1. Each component has an *event queue*, to the end of which the component's ESM can push (i.e., add) events, and from the front of which it can pop (i.e., remove and process) events.
2. When a b-thread enters a new state, the execution infrastructure determines whether or not it is an *inter-component decision point* (ICDP), i.e., whether or not it should induce synchronization with certain other components (ICDPs are defined in the next subsection).
3. When a component's ESM receives an event that was broadcast by another component, the event is pushed to the end of the component's event queue.
4. When a component enters a new state (either initially, or following re-synchronization of all b-threads following the triggering of an event that affected at least one b-thread), the ESM does the following:
   (a) If the component's event queue has at least one event, the ESM pops the first event from the queue, and triggers it (i.e., notifies affected b-threads, who then change states and re-synchronize).
   (b) If the queue is empty, then
      i. If one of the b-threads is in an ICDP, the ESM waits for the components specified in the ICDP to reach the corresponding ICDP and/or confirms that they are already at that point (note that no component goes past an ICDP without synchronizing with the others). If two ICDPs are in effect concurrently, they are handled, separately, in arbitrary order. Hence, all the components involved in an inter-component decision consider the same sets of requested and blocked events. The components then synchronize and mutually agree upon the triggered event. This event is then broadcast to all components (including the ones involved in the decision itself). Note that the chosen event may or may not be one of those that induced the need for inter-component decision. In the latter case, the b-threads that induced the ICDP will not react to the chosen event, and the component will be at an ICDP at the next synchronization point as well.
      ii. If there is an event that is in the local requested event set and not in the local blocked event set (for the current composite, synchronized state of all b-threads in the specification as modified locally under R&P), the ESM triggers that event, and broadcasts it to all other components.
      iii. If the event queue is empty and there is no event that is locally enabled, the ESM waits for external events to arrive via broadcast from other components.

iv. Otherwise, that is, if the event queue is empty and there is no event that is locally enabled, the ESM waits for external events to arrive via broadcast from other components.

5. When b-threads are notified of selected events they change their states according to their local state-transition function (which is identical in all components and is the same as in the original non-distributed specification).

We observe that deadlock-detection needs to be treated differently in the distributed case compared to the centralized case. According to the semantics given in Section 2, the system can detect a deadlock if the ESM determines at some point that all requested events are blocked, so that none can be selected. This, of course, holds only in the case of static scenarios, and where simulation of environment behavior is already included in the model. By contrast, in the distributed case this is no longer the case, as components begin to serve as each other's environment: If one of the local b-threads waits for an event that is external to the component, another component might broadcast that event. Thus, the component should just be stalled until such a broadcast arrives.

Definition 2. *A* distributed model *produced from a behavioral model M, with respect to a strict event separation, $S = \{E_1, \ldots, E_k\}$, denoted as $\mathcal{D}(M,S)$, is defined to be the set of projections of M along the components of the event separation:*

$$\mathcal{D}(M,S) = \{project(M,E_1), \ldots, project(M,E_k)\}.$$

Executing a distributed model means executing the component models (i.e., the projections) according to the operational semantics defined in this section.

5.4 Conditions for Inter-component Synchronization

The following definition is useful in identifying the points during the execution in which multiple components need to synchronize:

Definition 3. *Given a component model $C_j = project(M,E_j)$, a b-thread BT^i and some state $q \in Q^i$. We say that BT^i is* controlled *by C_j at state q if one or more of E_j's local events is requested or waited-for in q; i.e., if $\exists e \in E_j$ such that $\delta^i(q,e) \neq q$ or $e \in R^i(q)$.*

Definition 4. *Given a component model $C_j = project(M,E_j)$, we call a state $q \in Q^i$ in a component's b-thread BT^i an* inter-component decision point (ICDP) *if and only if q is controlled by multiple components and $\exists e \in E_j$ such that $e \in R^i(q)$.*

The R&P distribution process dynamically determines when a b-thread is in a state that is an inter-component decision point per Definition 4.

For example, assume that in the original specification for a four-wheel vehicle a single b-thread requests two events (e.g., steerRight and steerLeft), allowing the ESM to non-deterministically choose one, as would be the case if a 'random walk' were desired. Then, in the distributed implementation, if the two events end up in a single physical component, this will not be an ICDP. But, if they are in separate components, coordination will be required, naturally, and this will be an ICDP. Consider also the case where these two events are requested by two separate b-threads. In a centralized

implementation this will be valid, especially if each of the two b-threads also waits at this point for the other b-thread's event and stops requesting its own if it sees that the other's request is selected. Moreover, if the two events are in distinct components as before, then the requesting and waiting (in a single b-thread) would cause the corresponding state, which appears in the replicated b-thread in both components, to be marked as an ICDP, yielding the same sets of runs. Alternatively, if the events are indeed in physically independent components, as would be the case when steerRight is implemented by advancing (rolling forward) the left front wheel, (and steerLeft, respectively, by advancing the right wheel), then the developer has the option of removing the waiting-for-the-other-event from the bSync call in that state. In this case, these states will no longer be ICDPs, and one of the possible runs is that both requested events will be selected (one after the other), both front wheels will be advanced, and the vehicle will advance forward rather than turn. We note however, that here the specification and the set of runs has changed dramatically to accommodate, or take advantage of, some new capabilities of the distributed environment, and we no longer attempt to preserve the set of original runs.

It is important to note that the properties that induce the existence of an ICDP are properties of a single state of a single b-thread and not of the entire specification: the set of a all b-threads may, at the same time (i.e., at a given synchronization point), request and/or wait for events controlled by multiple components, but if no single b-thread is controlled by two components, this will not force an inter-component decision. However, at any synchronization point in any component (which means synchronization of all b-threads in that component), if a single b-thread is in an ICDP, the ESM will synchronize the entire component with the other affected components.

When at an ICDP, the actual joint decision of multiple, already-synchonized ESMs, can be performed, e.g., via a distributed leader election protocol [10]. Once a specific ESM is selected as the leader, it chooses the next triggered event based on the local requested and blocked events in its current state.

Note that Definition 4 mandates that the requested-events set not be empty. This restriction reduces the scope of when an ICDP is called for. Consider for example a logger scenario that, obliviously to any synchronization implications, waits for all possible events in the system and writes the relevant data to a log file (without requesting any behavioral event). Without the requirement that at least one event be requested by the b-thread causing the ICDP, such a simple logger would cause the entire execution to synchronize at every event selection. However, this feature, which enables more asynchronous execution, has its price. E.g., two such simple logger scenarios running in two separate components may observe differently the order of a given sequence of events. The issue of seeing different orders may be resolved either by programming the application such that it induces an ICDP only when it is called for explicitly by the requirements, or by order-enforcing infrastructure, such as the one as described below in Subsection 5.5 and Assumption 1.

5.5 Equivalence to Centralized Executions

As described above, given a centralized behavioral model M over an event set Σ and a strict event separation $\{E_1, \ldots, E_k\}$, our distribution process produces a set of compo-

nent models $\{C_1, \ldots, C_k\}$, whose execution then follows a very particular protocol. We would like to prove that, under certain assumptions, this distributed model behaves like the centralized model, i.e. the set of all possible executions of the distributed model is identical to those of the centralized model.

First, we present the following assumptions.

Assumption 1 (Strict and total event ordering). Given $\mathcal{D}(M,S)$ (Definition 2), we assume that there exists a strict total ordering of all selected events, and this ordering is global and visible to all components (see Section 4). I.e., for any pair of events a,b selected by any one or two components, exactly one of the following is true:

- a happened before b, or
- b happened before a

and, all components observe these events in the same order.

Stating the above more formally, we assume that in each component model $C_i = \mathit{project}(M,E_i)$, the event queue described in the R&P execution semantics is subsumed by a virtual queue, termed VQueue and denoted $\hat{Q}_i$, with the following properties as well as communication and execution semantics. After an event e is selected by a component C, the event is pushed atomically and simultaneously onto all VQueues of all components (including the one where it was selected). Notice that the atomicity here regards all pushes onto all queues, and any event selection or other important behavioral processing action (including another collective push) occurs either before or after such a collective-push action of one selected event. Each component processes events by popping them, one at a time, from its VQueue, and announcing the event to the b-threads running in that component (which are, in fact, all the b-threads in the specification, as modified/projected locally by R&P). The b-threads then change states according to BP semantics and resynchronize locally. The next event selection at this component can occur at any time during this process as long as all events previously selected *by this component* (and pushed onto its VQueue and onto the VQueues of all the other components), have been popped from the local VQueue $\hat{Q}_i$ and fully processed. However, the local VQueue does not need to be empty when the event selection occurs, i.e., it may contain events that were pushed onto it by other components, since the previous local event selection.

One may consider this assumption as limiting the class of applications covered by the formal argument to those where notification of events to components is serialized by some virtual central controller, and, where each component waits for the arrival of all events that were triggered in any component after its own last event selection, before the next event selection. In Sections 8 and 9 we discuss why these limitations are not of great concern and do not diminish from the power of R&P and of reduction of synchronization requirements.

Assumption 2 (No reliance on cross-component blocking of already-enabled events). Let $\mathcal{D}(M,S)$ be a distributed model that is being executed. For a given component C_j, let $\hat{Q}_i$ be the totally ordered set $\{e_1, e_2, \ldots e_m\}$, i.e., these are the pending events in its VQueue. Let $r = \langle q^1, \ldots, q^n \rangle$ be the component's current program cut (Definition 1).

The component's enabled events are:

$$E_r = \left(\bigcup_i R_j^i(q^i) \right) \setminus \left(\bigcup_i B_j^i(q^i) \right)$$

We assume that popping events from the queue does not remove elements from E_r, i.e., $\forall l \leq m$, Let $\dot{q}_l^i = \delta^i(\ldots\delta^i(\delta^i(q^i,e_1),e_2)\ldots,e_l)$ and

$$E_r \bigcap \left(\bigcup_i B_j^i(\dot{q}_l^i) \right) = \emptyset \tag{1}$$

This is in line with the discussion of not relying on cross-component blocking of already-enabled events in Subsection 4.2. In other words, we assume blocking is done sufficiently in advance to avoid race conditions.

Lemma 1. *Under Assumptions 1 and 2, the set of all possible executions (the* language*) of M is identical to the set of all possible executions produced by the component models* $\{C_1,\ldots,C_k\}$ *when run jointly in a distributed fashion.*

This lemma, which is the main proven result of this work, is of practical importance, as it implies that the proposed R&P distribution process will not cause the model to behave in unexpected ways. As discussed under the Language Equivalence principle in Subsection 4.1, note that the lemma is about the collection of all runs, and *does not* claim that if the distributed and centralized models are run side-by-side, they will always produce the same run. The main reason is that in a cut where more than one event is enabled, we cannot guarantee that two side-by-side runs of the executable specification will make the same choice; and, this holds independently of whether either of them is centralized or distributed. Given the language equivalence result, one can study and analyze the centralized version of the model (which is far easier for humans to grasp and comprehend, and for tools to analyze) and the conclusions will apply to the distributed setting too. We will discuss some of the implications of this result in Section 9.

Note that for the lemma to hold we also implicitly assume that each enabled event has a positive probability of being selected. If the event selection is *unfair*, in the sense that it always selects certain events and not others in particular situations, then the lemma will not hold. We do not consider this assumption to be a major constraint on the kind of applications supported by R&P.

Proof of Lemma 1: Assumption 1 and Assumption 2 shape the rest of the proof. Components select events based on standard BP execution semantics applied to the replicated-and-projected specification. Those selected events are immediately pushed into all the event queues of all components. This operation is instantaneous and defines some global order among the selected events. We do not define when components pop, announce, and process events from their event queue, but simply assume that they do so at some point, and soon enough as to not violate assumption 2.

Claim 1. In a distributed execution of $\mathcal{D}(M,S)$, if at any point in time all components empty their event queues $\hat{Q}_i$ (processing the events), then the cuts of all component models are at the same state.

Proof. Given a component model $C_i = project(M, E_i)$ and its event queue $\hat{Q}_i$, let $\{e_{l_1}, e_{l_2}, \ldots, e_{l_m}\}$ be all the events, in order, popped from the queue and processed by the component since the execution started. By Assumption 1, the indices $\{l_1, l_2, \ldots\}$ are identical for all components. And, since we assume that selected events are pushed into all event queues instantly, once components empty their queue the total count of processed events is also the same for all components.

While it may be obvious that at any instance at most one event will be selected, in exactly one component (and all components will eventually see this event), when considering possible causes of divergence it is useful to notice that:

1. Given that components are, in general, not synchronized, their event selections are always strictly ordered. The event selection in one component is always before or after any event selection in any other component.
2. In a given cut in a given component, if (after R&P) multiple events are enabled, then:
   (a) If these enabled events are controlled by this same component, then this is the only component in which they can be enabled. The one event that will be chosen by this component from this set will be visible identically to all components.
   (b) If the enabled events are controlled by multiple components, then the cut meets the requirements for ICDP, and all the relevant components are also synchronized at the cut at hand; a single event will be chosen via a an agreed-upon decision, made for all of them.

Therefore, all components process the same totally ordered set of events $\{e_{l_1}, e_{l_2}, \ldots, e_{l_m}\}$.

Observe that in the execution of $\mathcal{D}(M,S)$ all components begin at the same initial program cut $\langle q_0^1, \ldots, q_0^n \rangle$, and after m steps a projected b-thread BT_j^i in component C_j transitions to some state $\dot{q}_m^i = \delta^i(\ldots\delta^i(\delta^i(q_0^i, e_{l_1}), e_{l_2})\ldots, e_{l_m})$. By definition of the projection process, the δ^i functions are identical across components, and hence all projections of each thread proceed to the same state, $\forall i, r : \dot{q}_m^r = \dot{q}_m^i$. Therefore all component end up in the same cut. The claim follows. □

Corollary 2. *Given a distributed model $\mathcal{D}(M,S)$, all the components process the same totally ordered set of events.*

Proof. Follows immediately from claim 1. □

Using corollary 2 we can talk about the sequence of events processed by $\mathcal{D}(M,S)$, as all its components process the same sequence (albeit they might do so at different speeds).

We now define what the formal language generated by a behavioral model is, and prove that the languages of the distributed model and of the undistributed model are the same.

Recall that for an undistributed model M an *enabled event* at some program cut is an event that is requested by some b-thread and is not blocked by any of the b-threads. Recall also that under R&P all components run all b-threads but requesting and blocking of events take place only in components that control these events. We thus extend the *enabled event* term to a distributed system $\mathcal{D}(M,S)$ as follows:

Definition 5. *In a distributed model $\mathcal{D}(M,S)$, an* enabled event *is one that is requested by some b-thread of some component in which all b-threads are presently synchronized (i.e., a component that is at a cut), and, is not blocked by any b-thread in that component.*

Definition 6. *Let $\Delta(r,e)$ denote the program cut transition function, where r is a program cut and $e \in \Sigma$ is an event. Δ is fully defined by the b-threads state transition function δ^i as follows: for $r = \langle q^1,\ldots,q^n\rangle, \Delta(r,e) = \langle \delta^i(q^1,e),\ldots,\delta^i(q^n,e)\rangle$.*

Definition 7. *The* language *L of a behavioral model M denoted $L(M)$ is a set of words defined over the alphabet Σ. A word $w = e_1e_2\ldots$ is in $L(M)$ if its letters constitute a legal run of M; i.e., if we begin in the initial cut and apply Δ according to the sequence of events in w, the next event is always enabled in the current cut.*

The language *of the distributed model $\mathcal{D}(M)$ is defined similarly. A word w is in $L(\mathcal{D}(M,S))$ if and only if there exists a run of $L(\mathcal{D}(M,S))$ where the components select the totally ordered set of events in w. (We assume that the environment is incorporated into the behavioral model as b-threads that non-deterministically request environment events.)*

The equality between $L(M)$ and $L(\mathcal{D}(M,S))$ will follow from the following claim:

Claim 2. $L(\mathcal{D}(M,S)) \subseteq L(M)$

Proof. At any time during the execution of distributed model $\mathcal{D}(M,S)$, the enabled events are as determined by the cuts of those components that are presently in a cut (in fact, one can also conveniently assume that a cut transition in a component is always atomic in the sense that a component can only be observed when in a cut, yet not all components may be in the same cut at the same instance of time). As components cannot block external events, the set of enabled events at a given instance is the union of sets of enabled events of all components which are in a cut.

We will denote by E_m^M the set of enabled events of a centralized model after selecting $m \geq 0$ events. Likewise, we denote $E_m^{D_j}$ the set of enabled events of component C_j in the distributed model $\mathcal{D}(M,S)$ after m events have occurred. We do not specify the number of events $l \leq m$ that were actually popped and processed by the component. $E_m^D = \cup_j E_m^{D_j}$ is defined as the set of enabled events in the distributed model after m events were selected and each component C_j has processed $l_j \leq m$ events. By definition, the set of initial enabled events of M is

$$E_0^M = (\bigcup_i R^i(q_0^i)) \setminus (\bigcup_i B^i(q_0^i)) \tag{2}$$

and after m steps the set of enabled events of M is

$$E_m^M = (\bigcup_i R^i(q^i)) \setminus (\bigcup_i B^i(q^i)) \tag{3}$$

where $\langle q^1,\ldots,q^n\rangle$ is M's program cut after m events have occurred.

The set of initial enabled events in the distributed model $\mathcal{D}(M,S)$ is E_0^D. Clearly $E_0^D = E_0^M$.

Consider the distributed model $\mathcal{D}(M,S)$ after m steps, i.e., after selection and VQueue-ing of exactly m events as counted collectively in the entire model, and examine an arbitrary component C_j. The component has processed l events, where $l \leq m$, and has $m-l$ events in its VQueue. Specifically, it has processed the sequence of events defined by the totally ordered set $\{e_1, e_2, \ldots, e_l\}$, and $\hat{Q}_j = \{e_{l+1}, \ldots, e_m\}$.

Let $r = \langle q_j^1, \ldots, q_j^n \rangle$ be the current program cut of C_j and let ξ_l be the set of enabled events of C_j after processing $l \leq m$ events.

By Assumption 1, if $\hat{Q}_j$ contains an event selected by C_j then the component will not attempt to select another event, until processing that event, and therefore, effectively, $\xi_l = \emptyset$. Otherwise, at this stage the set of enabled events is defined by:

$$\xi_l = \left[\left(\bigcup_i R^i(q^i) \right) \setminus \left(\bigcup_i B^i(q^i) \right) \right] \cap C_j = \left(\bigcup_i R_j^i(q_j^i) \right) \setminus \left(\bigcup_i B_j^i(q_j^i) \right). \tag{4}$$

Whenever C_j processes any number of the $m-l$ events in $\hat{Q}_j$, no enabled events will removed from ξ_l for the following reasons:

- No b-thread will change into a state where it blocks events in $E_l^{D_j}$. This is due to Assumption 2 which claims that the application does not rely on instantaneous blocking.
- No b-thread that requested an event will change into a state where it no longer requests this event as this would imply that this b-thread was simultaneously requesting a local event and waiting for an external one which would then require an ICDP. As discussed before, an event chosen by an inter-component decision is considered as selected by all participating components, and we had assumed that $\hat{Q}_j$ contains no event selected by C_j.

Therefore

$$\forall 0 \leq l \leq m : \xi_l \subseteq \xi_m.$$

By definition, ξ_m is the set of enabled events in component C_j after processing all the m events from the VQueue, therefore, as the VQueue is empty, ξ_m is simply:

$$\xi_m = \left[\left(\bigcup_i R^i(q^i) \right) \setminus \left(\bigcup_i B^i(q^i) \right) \right] \cap C_j \subseteq \left(\bigcup_i R^i(q^i) \right) \setminus \left(\bigcup_i B^i(q^i) \right) = E_m^M$$

By the way we defined $E_m^{D_j}$ the following holds: $E_m^{D_j} = \xi_l$ for some $l \leq m$, but as we saw $\forall 0 \leq l \leq m : \xi_l \subseteq E_m^M$, therefore $E_m^{D_j} \subseteq E_m^M$ and the following holds:

$$\forall m \geq 0 : E_m^D = \cup_j E_m^{D_j} \subseteq E_m^M.$$

Therefore $L(\mathcal{D}(M,S)) \subseteq L(M)$. □

Claim 3. The language of a behavioral model $L(M)$ is equal to the language of its distributed version $L(\mathcal{D}(M,S))$.

Proof. We need to show that $L(M) \subseteq L(\mathcal{D}(M,S))$. That is trivial: Assume that a run of $\mathcal{D}(M,S)$ always empties its VQueues instantly as soon as events are pushed. In this case the distributed model would behave identically to the centralized version. Ergo $L(M) \subseteq L(\mathcal{D}(M,S))$.

As $L(\mathcal{D}(M,S)) \subseteq L(M)$ by claim 2 it immediately follows that

$$L(\mathcal{D}(M,S)) = L(M)$$

□

This concludes the proof of Lemma 1, which also implies that the distributed model *behaves correctly*, i.e., produces executions that are allowed under BP semantics.

6 Per-Component Timescales

As explained earlier, in a centralized behavioral model, all b-threads must synchronize in order for the ESM to announce the selected event. The b-thread that takes the longest to reach its synchronization point (e.g., because it performs slow local calculations or writes to a file) forces the rest of the b-threads to wait until it synchronizes. This lockstep execution thus results in the slowest b-thread dictating the timescale for the whole system. This is a common issue in behavioral models that involve multiple scenarios operating on different timescales (see, e.g., [17]), and it also applies to our distributed variant of BP: for example, a slower component might experience delays before broadcasting events that a faster component depends on, forcing the latter to wait. Furthermore, external events can "pile up", increasing the processing time of future event selections and delaying the selection of potentially crucial events.

In this section we discuss how to allow the generated components to operate efficiently on different timescales.

Previous work [17] has tackled this difficulty in a variety of ways. One approach in [17] introduced an *eager execution* mechanism for behavioral models. This technique lessened the severity of the problem by sometimes allowing the ESM to trigger an event even when some of the b-threads have not yet synchronized. Our distribution technique lends itself naturally to this kind of idea, because within a given component, we know that b-threads controlled by other components, which have not synchonized yet, cannot block local requested events. Thus, by applying a method similar to eager execution, the ESM does not have to wait for b-threads which wait only for external events (such

b-threads may be in the original specification, or they may be the projected version of b-threads with event requests changed to waiting for events).

In our distributed setting, eager execution can be applied as follows. Given a behavioral model $M = \{BT^1, \ldots, BT^n\}$ and its distributed component models $\{C_1, \ldots, C_k\}$, let $q \in Q^i$ be a state in which b-thread BT^i is not controlled by component C_j. Observe BT^i_j, i.e., the copy of BT^i that is running in component C_j. Because BT^i_j is not controlled by C_j, it does not request or wait for any local events and must be waiting for an external event e controlled by some other component C_m. In other words, until such time as e is triggered by C_m, thread BT^i_j will not affect local event selection in component C_j. In such situations we propose to temporarily *detach* thread BT^i_j from its local ESM, effectively allowing event selection in component C_j without considering BT^i_j. This allows component C_j to operate in its own pace, while BT^i_j can be regarded as temporarily operating in the same time scale as C_m. Whenever e is finally triggered and BT^i_j reaches a new state $\bar{q}$ in which it is controlled by C_j, it is reattached to the local ESM. This technique readily enables different components to simultaneously operate at different timescales.

To support eager execution within our distributed framework, the external event queue within each component model needs to be decoupled from the distributed ESM. Instead, each b-thread in the component receives its own external-event queue, and at each synchronization point pops all external events and selects them one at a time. The changes in the BP execution engine are summarized as follows:

- Each b-thread should flag itself as synchronized or unsynchronized with each bSync call, depending on the state.
- A separate event queue is created in each b-thread, thus allowing b-threads to process external events independently of the local ESM. A b-thread that arrives at a state first empties its event queue by repeatedly popping and selecting an event.
- External events received at a given component are injected into all the b-thread event queues by the component's BP execution engine. B-threads that are already awaiting the local ESM are notified to handle the external events.

7 Example and Evaluation

We now describe in more detail the distributed application upon which we carried out our evaluation. Specifically, and as introduced in Section 3, we implement a drone-based light show as follows. Each of four drones has a green light and a red light. Initially, the drones "do the wave", each flashing its green light briefly, in turn. This is implemented by the scenario in Algorithm 1. The scenario in Algorithm 2 shows the projection of the scenario in Algorithm 1 to Drone1.

Our example is a slightly richer scenario, coded as a behavioral program written in C++. The four drones (labeled Drone0 through Drone3) participate in "a green wave", starting with Drone0. After the conclusion of two full cycles, the drones jointly decide which of the drones will start the next wave. The next wave will, again, last for two full cycles, and the entire process repeats five times. For now, the entire specification consists of a single scenario. In this implementation, the light-flashing events are labeled as FlashGreen0 through FlashGreen3, each representing the flashing of the light

```
i=0;
while true do
    bSync(R = {FlashGreen((0+i)%4)});
    bSync(R = {FlashGreen((1+i)%4)});
    bSync(R = {FlashGreen((2+i)%4)});
    bSync(R = {FlashGreen((3+i)%4)});
    nextEvent = bSync(R = {NW0, NW1, NW2, NW3});
    i = indexOfWave(nextEvent);
end
```

Algorithm 1: Pseudocode of a BP scenario demonstrating a simple undistributed wave example. For each bSync synchronization point, R is set requested events. The events NW*0* through NW*3* indicate a request the start a new wave at the corresponding component. These events are requested after each full cycle, and BP event selection then decides which component starts the new wave. The method *indexOfWave* translates an event NW*i* to the index *i*.

```
i=0;
while true do
    bSync(W = {FlashGreen((0+i)%4)});
    bSync(R = {FlashGreen((1+i)%4)});
    bSync(W = {FlashGreen((2+i)%4)});
    bSync(W = {FlashGreen((3+i)%4)});
    nextEvent = bSync(R = {NW1}, W = {NW0, NW2, NW3});
    i = indexOfWave(nextEvent);
end
```

Algorithm 2: Projection of the scenario of Algorithm 1 onto the component Drone1. Notice that requested events controlled by other components become waited-for (represented by the W sets).

in the respective drone, in either a centralized or distributed implementation. The selection of the drone that will start the next wave is carried out by the scenarios requesting four "new wave" events, NW*0* through NW*3*, and the BP event-selection mechanism arbitrarily selecting one of these events. We then associate each of the FlashGreen and the NW events with the corresponding component. In this simplified example the duration of the flashing of each light is implemented in a delay (sleep) of 250 msec in the b-thread that is about the request a FlashGreen event.

For simplicity, this implementation uses a *centralizer* component and does not implement a leader-election mechanism. The centralizer is an infrastructure component which is responsbile for: (i) receiving notifications of events triggered in any behavior components, and broadcasting this information to all other components, and (ii) managing joint decisions, by receiving notices from any component ESM that wishes to synchronize, which include the sets of requested and blocked events, waiting for all other components to reach their corresponding state, selecting an event which is requested and not blocked, and notifying all components of the selection. Note that the centralizer serves only in simulations and studies of the approach, and that in real distributed implementations broadcasting can be performed by a vartiety of techniques (including the above), and joint decisions can be reached by classical distributed-processing solutions, such as leader election.

At this point it is important to distinguish between the concepts of classes and objects and the concept of components as used here. Events may be self-standing entities, or they may be associated with objects. In our example, each drone is a component, and objects may reside within a component, or may span multiple component. Such objects can be, e.g., a drone controller, a drone light, a wave effect (which can have a beginning and end events, or a color property) or an entire light show. As can be seen in the example given in Algorithm 2, each component executes "the entire specification", in this case, this one scenario. In the distributed implementation, when scenarios request or wait for FlashGreen events, they do not synchronize, but when they request the four new wave events, they all synchronize. This results in a partially synchronized execution, which mimics the centralized execution but does so with less inter-component synchronization.

We compare our target, partially synchronized execution of a specification created with the replicate-and-project implementation (R&P), with a fully synchronized distributed execution (abbr. FS), where each component executes the same specification, and they synchronize with every event selection. The decision in each component whether to actually turn on its own light following its respective FlashGreen event is left as a small implementation detail, i.e., the light-switch actuation method skips the operation if there is no direct connection with the device. Both implementations execute the same one-scenario specification, replicated over four components. The total number of events that occurred, all of which were broadcast to all components, is 44 — the same for FS and for R&P (five repetitions of two four-event cycles, and four joint decisions). In the R&P however, only four of these required synchronization. The total execution time was the same in both cases, dominated by the duration of the light flashes, but if synchronization delay is artificially increased, total execution time is increased accordingly (e.g., a 100 msec delay purely due to synchronization, in addition to any ordinary communication delay, would add 400 msec to the duration of each cycle of this single wave).

We now extend our mini-light-show example with another wave of flashing lights. We add a scenario in which, starting with Drone2, each of the drones briefly flashes a red light, in its turn. This multi-cycle wave continues uninterrupted and with no change until the ten cycles of the green wave terminate. The delay (sleep) before requesting a FlashRed event is 1000 msec. When multiple events are requested e.g., both a FlashRed together with FlashGreen or NW, the ESM selects an event at random. The forty FlashGreen events in the ten-cycles determine the beginning and end of the run, and the number of FlashRed events selected during this time varies. Since we are presently more interested in understanding the underlying effects than in measuring improvements over a large number of runs, we suffice with this artificial example. To highlight these effects we show in Table 1 a comparison of the two cases when in both FS and R&P, 44 FlashGreen events were triggered.

The basic communication delay in these experiments is set to 50 msec, resulting in 100 msec delay for broadcasting an event occurence via the centralizer.

Some interesting explanations and observations include:

- In FS, at every synhcronization point, both a FlashRed event, and, either a FlashGreen or NW events are enabled. This is true regardless of sleep delays and number

of components. Hence in such runs, on average, half of the events will be FlashRed. By contrast in R&P, FlashRed is enabled in a component together with one of the other two events in a way that depends on lengths of sleep delays and on the number of components in the cycle, yielding, in our case fewer FlashRed events during the run.
- Common to all runs is a $40 * 250$ msec taken by the FlashGreen events, plus $4 * 100$ msec minimum number of joint decisions, plus about 3 seconds of overhead (total of 13-14 seconds).
- The 41 seconds duration of R&P is the result of adding to the above ~13 seconds $28 * 1000$ msec FlashRed events.
- The 67 seconds duration of FS is the result of adding to the above 41 seconds of R&P $17 * 1000$ msec of additional FlashRed events and $85 * 100$ msec communication delays due the additional synchronizations, all of which had to occur during the same ten cycles of the green wave.
- Even though the total number of events triggered in R&P is less than in FS, the per-second event rate is higher.
- In the worst case, the performance of a distributed system resulting from an R&P distribution process will be the same as when a replicated specification executes without local changes in all components, with full synchronization at every event selection.

Table 1. Comparing an execution of a fully synchronized (FS) implementation of a two-scenario specification in a four-component configuration, to an execution of the partially synchronized replicate-and-project implementation (R&P). See discussion in the Section 7.

Measure:	FS	R&P
Number of FlashGreen event notification broadcast	40	40
Number of FlashRed event notification broadcast*	45	28
Number of "new wave" event notification broadcast	4	4
Total number of events	89	72
Total number of Inter-component synchronizations	89	4
Run duration (in seconds)	67	41
Events per second	1.32	1.75

While the above examples illustrate and quantify the kind of savings resulting from reduced synchronization, we must note that the synchronization delay itself is sometimes not the main issue. For example, if we were to replace the FlashGreen event(s) in our design with, e.g., pairs of TurnGreenLightOn and TurnGreenLightOff events, all scenarios might have had enough time to synchronize with each other following the event TurnGreenLightOn, in parallel to waiting for the time ticks that would signal the end of the shining of the light. A relaxed synchronization approach, separating the scenarios of the two waves into separate modules within each component, would further streamline an otherwise fully synchronized implementation. Nevertheless, the reduced inter-component synchronization still helps in simplifying the designs, and in enhancing system robustness. For example, consider recovering from loss of a drone, due to

battery running out, while “the show must go on”. It is much easier for all drones to observe and react to delays in other drones’ behavior, when they are fully functional as opposed to waiting in a global synchronization point (even when the latter is enhanced with timeout facilities as in [18]).

8 Related Work and Comparison

Distributed system have been the subject of extensive research and studies; see, e.g., [3, 32]. In general most approaches that aim to distribute a centralized system fall into one of the following classes:

1. The distribution process employs a kind of orthogonalization (or partitioning) process that decomposes the system into independent, orthogonal partitions that form the distributed system. This might be done with some user intervention and input or using a fully automatic process. The resulting executable partitions model parts of the system which can be ran, in parallel, as a distributed system without ever requiring to synchronize. Typical examples include the parallelization of an abstract computation, or the execution of a multi-agent system where an agent may wait for another agent’s messages, or may even coordinate a joint application decision, but they cannot in any way depend on synchronizing with each other their own internal computations and processes.
2. The executable partitions that form the distributed system are given in advance, and they do not map to logically-orthogonal parts of the specification. Instead, they are formed to satisfy other constraints (physical properties, performance, etc.) Unlike systems with orthogonal partitioning, some synchronization might be required to ensure the distributed system has largely the same function as the original one. A typical example would be a distributed database whose components are defined by physical machine capacities and boundaries. The component synchronize as part of their underlying computation, to ensure properties such as atomicity, consistency, isolation and durability (ACID).

Each class has a unique difficulty: Orthogonalization or synchronization. In terms of system design synchronizing distributed systems enjoy a larger degree of freedom in the way the distributed partitions can be chosen. Behavioral specifications generally do not expose orthogonal partitions that map to the physical parts or properties of the system, or at times, none at all. Performance-wise, the trade-off between an orthogonal and a non-orthogonal approach can be seen as the trade-off between distribution performance as opposed to execution performance.

Within the realm of behavioral programming, the research in [28] suggests an approach for orthogonal distribution, where the distributed system consists of multiple, manually design, independent programs, termed *behavior nodes (b-nodes)*, each with its own set of internal events. As this is an orthogonal approach, those b-nodes never need to synchronize with each other. Similar to our approach the b-nodes communicate by external events, however those events require manual translation to and from internal events. While in [28] the distributed system is generated by a manual partitioning of a

model into multiple b-programs, [15] proposes a synchronizing approach for distributing BP models by manually partitioning the b-threads of a single b-program into modules, where each module runs its set of b-threads and synchronizes with other modules upon choosing events that might matter to other modules. The set of events that require synchronization as well as which modules each events needs to synchronize with is known a priori. The research in [28] and [15] contains examples of an orthogonal distribution approach and a synchronizing one, respectively, in behavioral programming. However, in both approaches the component structure is dynamic and implied by the specification, in contrast to the present paper where the component structure is dictated by the physical structure of the system and external events emerge naturally and automatically from internal events. Furthermore our approach supports more general designs, inter-component scenarios and fine-grained synchronizations when scenarios give rise to inter-component decisions.

A different framework for the distributed execution of scenarios is presented in [12]. The approach there is similar to the one in this paper in that the distributed components can each choose to execute events that they are responsible for, and selected events are broadcast to all other components. Further, a coordinator component in [12] forces the situation where, as in Assumption 1, all components observe a single event order. The main issues with this implementation relative to R&P are that it requires that individual scenarios are written to not have states where events of multiple components are enabled. By contrast, R&P automatically coordinates all components when reaching a state where a joint decision is required, and it allows components to advance asynchronously when possible, and in particular, after locally selecting an event. An advantage, though, of the implementation of always enforcing a common event order in [12] is that it avoids the risk of sensitivity to different event orders. While Lemma 1 relies on such enforcement for the proof, R&P in general allows also for applications that forego this requirement, and solve order-dependencies in application-specific means. However, we must note that the actual reliance in the implementation on a physical centralized coordinator for the entire distributed system carries many disadvantages both in performance and robustness. The introduction of single order assumption in the proof of Lemma 1, can be seen more as an abstraction — a requirement that is either guaranteed by some efficient and robust means or by application-specific properties.

A more general, automatic handling of event-order dependencies in R&P , and possible generation of additional ICDPs, is left for future research, e.g. using formal methods, as discussed in Section 4.

The research in [14] describes (though without an implementation) a mechanism for the distributed execution of scenarios with *dynamic role bindings*. There, synchronization is done only among relevant components, as determined dynamically.

There has also been work on synthesizing scarcely-synchronizing distributed controllers from scenario-based specifications [4]. Distributed finite automaton controllers can be synthesized from scenario specifications in a way that greatly reduces communication overhead compared to previous approaches, especially compared to the broadcasts of events as also suggested in this work. However, the synthesis procedure is computationally complex and does not scale well as specification and system size increase. In [9], the authors study a similar problem and present an approach for synthesizing

executable implementations from specifications given in a distributed variant of LSC, termed *dLSC*.

Another work related to distribution of centralized scenario-based models (but outside of the realm of BP) is [34], which presents a synchronizing approach for distributing workflow specifications. This work exposes domain-specific knowledge in order to be able to generate automatically distributed partitions and synchronization semantics such that the resulting distributed system preserves the execution semantics of the original centralized version.

Outside the scope of scenario-based modeling [7] is an example of distribution of systems modeled using Petri Nets, specifically High Level Timed Petri Nets (HLTPN). This research uses the orthogonalization approach where the HLTPN is decomposed into subnets connected by *shared places*, nodes that are common to multiple subnets. Arcs are not allowed to cross subnet boundaries, ensuring that the decomposing is an orthogonal partitioning of the net. The shared places can be seen as global memory, shared between multiple subnets, used to control firing of transitions, however there is no synchronization between the subnets.

Further non-scenario-based research discusses the trade-off between performance optimization and communication minimization in parallel and distributed settings has been studied extensively. These two conflicting goals are discussed, e.g., in [5, 39]. In [38] the author suggests imposing certain limitations on the communication between the components, thus allowing for execution-time optimization to be performed during compilation.

It is interesting to note that distribution approaches that rely on scenario-based specifications typically exploit the execution semantics of the modeling language to generate synchronized distributed systems. Meanwhile, non-scenario-based approaches generally employ a form of orthogonalization, and usually rely on domain-specific knowledge or on high-level temporal specifications to facilitate the distribution. As discussed above, orthogonalization approaches are more rigid and are not always possible, feasible or applicable. To our knowledge there is scarcely any research that involves generating a non-orthogonal distributed system from a generic non-scenario-based specifications. It appears that despite (and perhaps due to) the simplicity of behavioral programming and of scenario-based specifications in general, it is generally amenable automated decomposition and distribution.

9 Discussion and Future Work

Previous research on scenario based programming has shown the great importance of formal methods and tools in ensuring that the resulting models, composed of many individual scenarios, perform as intended as a whole. Past efforts have yielded a large portfolio of tools for model checking [24], automatic repair [21, 30] and compositional verification [31, 16], and have even indicated that scenario-based programming may be more amenable to formal analysis than other modeling approaches [22, 19].

Given the above, applying formal analysis in the distributed case seems even more vital, as distributed models are inherently more difficult for humans to comprehend than centralized ones. Fortunately, Lemma 1 enables us to immediately apply existing

tools in our setting. Because the centralized and distributed models present the same behavior, it is possible to apply existing approaches to the centralized version and use them to draw conclusions regarding the distributed case.

Future research on new applications of formal methods to distributed implementations can also distill situations and "critical states" where special handling is needed. E.g., identify when there are special dependencies on observing a particular event order, and devise solutions that are automatic, reduce synchronization, and reduce the need for a total strict event order, and where the equivalence of the resulting distributed execution to the centralized one can be proven. For example, in the present implementation, an application that waits for two events from two different components in any order, and then transitions into the same final state, depends on guaranteed event order, and/or on two ICDPs, where, in fact, neither is required. This research will also include proofs for correctness of different distribution procedures — e.g., that in cases where the application does not depend on particular event order, a particular distribution method which does not guarantee Assumption **??** still works correctly. For example, cars arriving at an intersection, each detecting all other cars in their environment, do not have to rely on observing identical event orders. We wish to devise a distribution methods for scenario-based specifications for handling such situations, and prove their correctness, namely, that the cars executing these scenarios in a distributed manner indeed cross the intersection safely.

Nonetheless, in a distributed environment there are some hazards that do not appear in the fully-synchronized model, and may thus be overlooked by existing tools:

- **Inter-component deadlock:** An inter-component deadlock occurs when a component *C* has no enabled local events that it can trigger, and is thus waiting for certain external event(s). However due to various reasons, these external events may never arrive. For example, the reason might be that another component is actually waiting for an event that *C* needs to trigger. Note that a situation where a component is waiting for events local to a crashed component is not an inter-component deadlock, but a soft deadlock, as restarting the failed component might resolve the issue.
- **External event queue overflow:** When a component repeatedly takes longer to process external events than it takes the other components to trigger and broadcast these events, could result in exceeding the memory available for the external event queue. An example of this could be a logger component that takes too long to post its log entries to a remote location.
- **Latency:** Communication delays can cause poorly-designed systems to exhibit undesired behavior. As we discussed in Section 5.5, Lemma 1 does not hold when latency is too high, and so such errors cannot be detected by existing tools.

We are working on extending the presently available techniques to handle the issues listed above. For instance, in the latency case an improved model-checking algorithm might simulate a realistic latency for external event communication, depending on the communication method used (e.g., wired communications over a local network will have a much lower latency than a satellite connection). We are also exploring the use of quantitative approaches to formal verification to attempt and derive bounds on the maximal size a queue can reach, given certain constraints on the broadcast and processing times of system components.

In the context of inter-component deadlock, one approach for recovering from component failure or missed messages could be adding state information to the external events, permitting components that missed a transition to "fast-forward" to the correct state in a scenario. Another direction could involve having multiple instances of critical components, for redundancy.

As an additional future work direction, we would like to study approaches to choosing a strict event separation. While the components are usually derived manually from physical system requirements, at times it might be desired to delineate their boundaries automatically based on other criteria. One approach is to use clustering algorithms that take as input a function f that assigns, for every two events $e_1, e_2 \in \Sigma$ a correlation value $f(e_1, e_2) \in [-1, +1]$. The clustering algorithms then attempt to partition the events into a strict separation into k components (with k either known or unknown beforehand), such that two events are in the same component if their correlation is high and are in separate components if their correlation is low. While this problem is known to be NP-Complete, it can be approximated up to a log-factor [2].

10 Conclusion

The replicate-and-project approach transforms a centralized scenario-based specification so that it can be executed in a distributed configuration, by creating component-specific variations, based on each component's capabilities. We have shown that the resulting distributed models behave similarly to the centralized model from which they originated. This important property allows us to carry out most of the modeling work, including testing and analysis, in the centralized setting, which is easier to model-check and reason about. The projected models retain the naturalness and incrementality traits of behavioral programming. In their avoidance of excessive synchronization, they improve robustness and the ability to model systems with multiple time scales. In addition to the advantages of this approach in streamlining design and improving performance, it captures the more general fact that distributed operations that are robust and efficient often involve the sharing of knowledge between components, such that each of them knows at least some of the rules that control the behavior of the others - a concept whose applicability me go beyond scenario-based / behavioral programming.

Acknowledgements

This work is funded by grants from the German-Israeli Foundation for Scientific Research and Development (GIF) and from the Israel Science Foundation (ISF).

References

1. G. Alexandron, M. Armoni, M. Gordon, and D. Harel. Scenario-Based Programming: Reducing the Cognitive Load, Fostering Abstract Thinking. In *Proc. 36th Int. Conf. on Software Engineering (ICSE)*, pages 311–320, 2014.
2. N. Bansal, A. Blum, and S. Chawla. Correlation Clustering. *Machine Learning*, 56(1–3):89–113, 2004.

3. Jacek Błażewicz, Klaus Ecker, Brigitte Plateau, and Denis Trystram. *Handbook on parallel and distributed processing*. Springer Science & Business Media, 2013.
4. C. Brenner, J. Greenyer, and W. Schäfer. On-the-Fly Synthesis of Scarcely Synchronizing Distributed Controllers from Scenario-Based Specifications. In *Proc. 18th Int. Conf. on Fundamental Approaches to Software Engineering (FASE)*, pages 51–65, 2015.
5. Y. Cheng and T. Robertazii. Distributed Computation with Communication Delay [Distributed Intelligent Sensor Networks]. *IEEE Transactions on Aerospace and Electronic Systems*, 24(6):700–712, 1988.
6. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
7. Juan A. De La Puente, Alejandro Alonso, Gonzalo León, and Juan Carlos Dueñas. Distributed execution of specifications. *Real-Time Systems*, 5(2):213–234, 1993.
8. P Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.
9. D. Fahland and A. Kantor. Synthesizing Decentralized Components from a Variant of Live Sequence Charts. In *Proc. 1st Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 25–38, 2013.
10. S. Ghosh and A. Gupta. An Exercise in Fault-containment: Self-stabilizing Leader Election. *Inf. Process. Lett.*, 59(5):281–288, 1996.
11. M. Gordon, A. Marron, and O. Meerbaum-Salant. Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming. In *Proc. 17th Conf. on Innovation and Technology in Computer Science Education (ITICSE)*, pages 198–203, 2012.
12. Joel Greenyer, Daniel Gritzner, Timo Gutjahr, Tim Duente, Stefan Dulle, Falk-David Deppe, Nils Glade, Marius Hilbich, Florian Koenig, Jannis Luennemann, Nils Prenner, Kevin Raetz, Thilo Schnelle, Martin Singer, Nicolas Tempelmeier, and Raphael Voges. Scenarios@run.time — Distributed Execution of Specifications on IoT-Connected Robots. In *Proc. 10th Int. Workshop on Models@Run.Time (MRT)*, pages 71–80, 2015.
13. Joel Greenyer, Daniel Gritzner, Guy Katz, and Assaf Marron. Scenario-Based Modeling and Synthesis for Reactive Systems with Dynamic System Structure in ScenarioTools. In *Proc. 19th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 16–32, 2016.
14. Joel Greenyer, Daniel Gritzner, Guy Katz, Assaf Marron, Nils Glade, Timo Gutjahr, and Florian König. Distributed Execution of Scenario-based Specifications of Structurally Dynamic Cyber-Physical Systems. In *Proc. 3rd Int. Conf. on System-Integrated Intelligence: New Challenges for Product and Production Engineering (SYSINT)*, pages 552–559, 2016.
15. D. Harel, A. Kantor, and G. Katz. Relaxing Synchronization Constraints in Behavioral Programs. In *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 355–372, 2013.
16. D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On Composing and Proving the Correctness of Reactive Behavior. In *Proc. 13th Int. Conf. on Embedded Software (EMSOFT)*, pages 1–10, 2013.
17. D. Harel, A. Kantor, G. Katz, A. Marron, G. Weiss, and G. Wiener. Towards Behavioral Programming in Distributed Architectures. *Science of Computer Programming*, 98(2):233–267, 2015.
18. D. Harel and G. Katz. Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures. In *Proc. 4th Int. Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!)*, pages 95–108, 2014.
19. D. Harel, G. Katz, R. Lampert, A. Marron, and G. Weiss. On the Succinctness of Idioms for Concurrent Programming. In *Proc. 26th Int. Conf. on Concurrency Theory (CONCUR)*, pages 85–99, 2015.

20. D. Harel, G. Katz, R. Marelly, and A. Marron. An Initial Wise Development Environment for Behavioral Models. In *Proc. 4th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 600–612, 2016.
21. D. Harel, G. Katz, A. Marron, and G. Weiss. Non-Intrusive Repair of Reactive Programs. In *Proc. 17th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 3–12, 2012.
22. D. Harel, G. Katz, A. Marron, and G. Weiss. The Effect of Concurrent Programming Idioms on Verification. In *Proc. 3rd Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 363–369, 2015.
23. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 378–398, 2002.
24. D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-Checking Behavioral Programs. In *Proc. 11th Int. Conf. on Embedded Software (EMSOFT)*, pages 279–288, 2011.
25. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
26. D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play In/Play-Out Approach. *Software and System Modeling (SoSyM)*, 2:82–107, 2003.
27. D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM*, 55(7):90–100, 2012.
28. D. Harel, A. Marron, G. Weiss, and G. Wiener. Behavioral Programming, Decentralized Control, and Multiple Time Scales. In *Proc. 1st SPLASH Workshop on Programming Systems, Languages, and Applications based on Agents, Actors, and Decentralized Control (AGERE!)*, pages 171–182, 2011.
29. D. Harel and I. Segall. Synthesis from live sequence chart specifications. *Computer System Sciences*, 2011. To appear.
30. G. Katz. On Module-Based Abstraction and Repair of Behavioral Programs. In *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 518–535, 2013.
31. G. Katz, C. Barrett, and D. Harel. Theory-Aided Model Checking of Concurrent Transition Systems. In *Proc. 15th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 81–88, 2015.
32. Jieyao Liu, Ejaz Ahmed, Muhammad Shiraz, Abdullah Gani, Rajkumar Buyya, and Ahsan Qureshi. Application partitioning algorithms in mobile cloud computing: Taxonomy, review and future directions. *Journal of Network and Computer Applications*, 48:99–117, 2015.
33. Chris Miller and Christian Poellabauer. A Decentralized Approach to Minimum-Energy Broadcasting in Static Ad Hoc Networks. In *Proc. 8th Int. Conf. on Ad-Hoc, Mobile and Wireless Networks (ADHOC-NOW)*, pages 298–311, 2009.
34. P. Muth, D. Wodtke, J. Weissenfels, A. K. Dittrich, and G. Weikum. From centralized workflow specification to distributed workflow execution, 1998.
35. P. Ramadge and W. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM J. on Control and Optimization*, 25(1):206–230, 1987.
36. A. Stefanescu, J. Esparza, and A. Muscholl. Synthesis of Distributed Algorithms Using Asynchronous Automata. In *Proc. 14th Int. Conf. on Concurrency Theory (CONCUR)*, pages 27–41, 2003.
37. S. Steinberg, J. Greenyer, D. Gritzner, D. Harel, G. Katz, and A. Marron. Distributing scenario-based models: A replicate-and-project approach. *5th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, 2017.
38. A. van Gemund. The Importance of Synchronization Structure in Parallel Program Optimization. In *Proc. 11th Int. Conf. on Supercomputing (ICS)*, pages 164–171, 1997.

39. J. Yook, D. Tilbury, and N. Soparkar. Trading Computation for Bandwidth: Reducing Communication in Distributed Control Systems Using State Estimators. *IEEE Transactions on Control Systems Technology*, 10(4):503–518, 2002.