Supporting Verification-Driven Incremental Distributed Design of Controllers

Claudio Menghi*, Paola Spoletini†, Marsha Chechik‡, and Carlo Ghezzi*
* Politecnico di Milano, Italy; Email: {claudio.menghi|carlo.ghezzi}@polimi.it
† Kennesaw State University, USA; Email: pspoleti@kennesaw.edu
‡ University of Toronto, Canada; Email: chechik@cs.toronto.edu

***Abstract*—Controllers are software systems that interact with the environment in a complex manner. Because of their size and complexity, they require incremental development, modular decomposition, and decentralized development, possibly involving multiple parties. This paper proposes a comprehensive verification-driven framework, which provides a complete support to designers during development. The framework supports hierarchical decomposition into sub-controllers through formal specification in terms of pre- and post-conditions as well as independent development, reuse, synthesis and verification of sub-controllers.**

***Keywords*-Incremental development; controllers; verification; specification; model checking; synthesis.**

I. Introduction

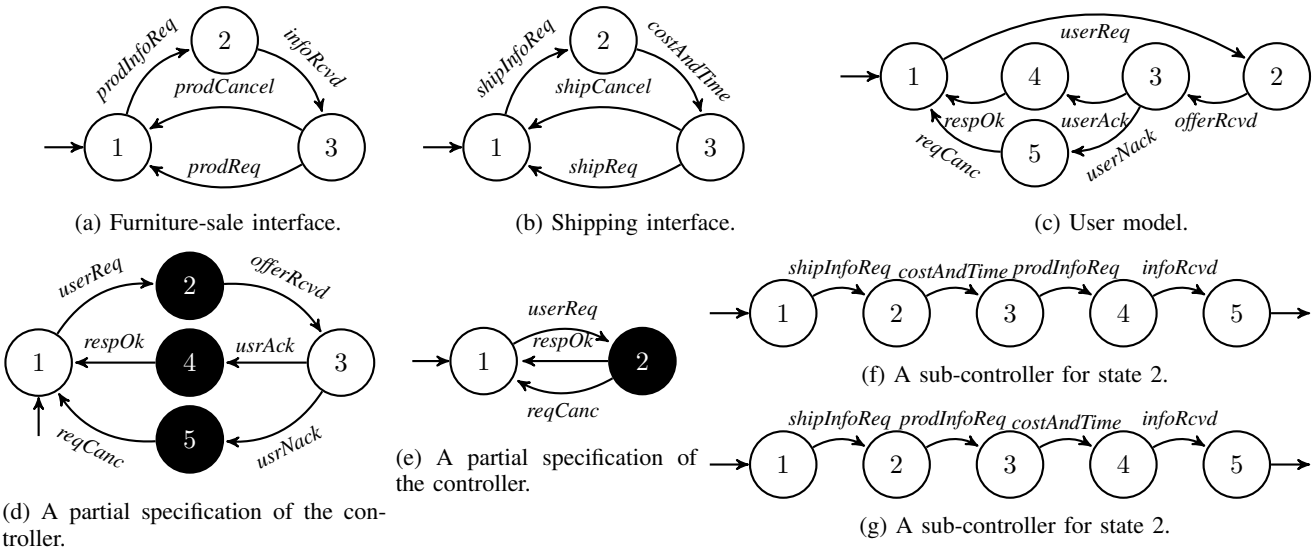
Controllers are software systems that interact with an external environment in a complex manner. Because of their criticality, size, and complexity, their design must follow a systematic approach, typically based on a recursive decomposition strategy that yields a modular structure. The decomposition identifies sub-controllers, which are given responsibility to control specific aspects of the environment. A good decomposition and a careful specification should allow sub-controllers to be developed in isolation by different members of the development team or be delegated to third parties [1]. It is also possible to reuse off-the-shelf components or delegate provision to external service providers.

Often controllers support safety-critical applications, whose overall correctness must be formally guaranteed. This is particularly challenging in the context of an incremental and distributed development strategy because of the intrinsic tension between the following two main requirements. On the one hand, to dominate complexity, we need to enable development of sub-controllers where only a partial view of the system is available [2]. On the other, we must ensure *a-priori* that the specifications provided for sub-controllers guarantee satisfaction of the global correctness properties of the controller once sub-controllers are independently verified to be correct. This means that at any stage of development, even when the design of the controller is still incomplete, we should be able to provide abstract specifications for the missing parts that both ensure correctness of the final controller and independent development of the parts. In conclusion, controller development should be supported by a process that: (1) is intrinsically iterative; (2) supports decentralized development; and (3) guarantees correctness at each development stage.

Recent research [3], [4] has focused on controller synthesis as a way to obtain correct-by-construction controllers. These techniques are tremendously useful for in-house development of small controllers or development of individual sub-controllers. Yet they are not appropriate for many real-world cases, due to their inability to support an incremental and distributed development, which becomes necessary when the problem is too large to be handled in a single step. In addition, they do not support reuse of portions of the controller. For these reasons, although significant steps have been made in the direction of synthesis, the dream of a fully automated approach based on synthesis is not yet and perhaps will never be a viable solution for complex controllers. A more viable solution is to use synthesis activities in the design process as a support to the human effort [5], [6].

The need for supporting incremental development of controllers has been recognized by others as well. For example, some approaches [3], [7] synthesized a partial model of a controller from properties and scenarios and then facilitated iterative development of this model through refinement. Others [8], [9] provided support for model-checking of partial models, with the goal of preserving correctness when such systems get refined. However, while being verification-driven, these techniques do not explicitly address the problem of distributed development and thus do not fully satisfy the requirements for supporting development of real-world controllers.

We believe that the complete automation of complex controller development is both impossible and undesired. Rather, the construction of the overall structure and the different development steps require insight and experience which are human, not machine characteristics [5]. This suggests the need for a framework that capitalizes on the synergy between humans and machines for supporting distributed incremental development by providing tools to *enhance (and verify) the human work at each step of the development*. In particular, the framework should comprise of the following activities: 1) Support for design activities performed by humans, whether it is to create an initial (partial) structure of the controller or a part of the controller, to specify properties, to model the environment interface, to fix problems, or to refine the system; 2) Support for verification via different tools to analyze the partial initial design, including support for model-checking it w.r.t. properties of interest; 3) Support for distributing



(a) Furniture-sale interface.

(b) Shipping interface.

(c) User model.

(d) A partial specification of the controller.

(e) A partial specification of the controller.

(f) A sub-controller for state 2.

(g) A sub-controller for state 2.

Fig. 1: Models of the p&d example.

development of not yet finished parts of the controller to other teams, including third-party vendors or service providers. Such development can be done manually, or it may be possible to reuse existing off-the-shelf components, or even automatically synthesize sub-controllers from specifications, if this is feasible. 4) Support for integrating the pieces contributed by distributed teams while guaranteeing that the resulting controller is still correct w.r.t. properties of interest.

In this paper, we describe a unified framework called FIDDLE (Framework for Iterative and Distributed Design of controLlErs), which supports controller development processes that follow the above overall strategy. FIDDLE supports the formal specification of global properties, the decomposition process and specification of sub-controller interfaces, and a set of tools to guarantee correctness of the different artifacts produced during the process. Although the main contribution of the paper is a method for supporting an iterative and distributed verification-driven controller development process through a coherent set of tools, specific novel contributions are: 1) definition of a new formalism, called Interface Partial Labelled Transition System (IPLTL), which can be used to specify a controller through a decomposition that encapsulates sub-controllers into unspecified black box states; 2) definition of a language to specify the expected behavior of a back-box state via pre- and post-conditions expressed in Fluent Linear Time Temporal Logic; 3) definition of the notion of a correct sub-controller and of a local verification procedure that guarantees preservation of global properties.

The approach has undergone a preliminary validation. First, we tried to assess its validity on a realistic case-study, which is based on reverse engineering the Executive module of the well-known Mars Rover system developed at NASA. Second, we generated random artificial examples of increasing complexity to demonstrate scalability of the approach.

The rest of the paper is organized as follows. Sec. II introduces a small motivating example. Sec. III provides a detailed overview of FIDDLE. Sec. IV provides the necessary background. Sec. V presents a novel modeling notation – Interface Partial Labelled Transition Systems (IPLTL) – and shows how to model-check and refine systems expressed in this notation. Sec. VI defines algorithms for automated components of FIDDLE and describes their implementation. Sec. VII reports on an evaluation of the effectiveness and scalability of the proposed approach. Sec. VIII compares FIDDLE with related approaches, and Sec. IX concludes the paper.

II. Motivating Example

Consider designing a controller for a simple *purchase&delivery* (p&d) example [10], [11]. The goal of the controller is to orchestrate the interaction between two Web services, called *furniture-sale* and *delivery-booking*, so that a set of provided safety and liveness requirements is satisfied.

In this example, the environment is comprised of three parts: the furniture-sale service, the delivery-booking service and the user. Fig. 1a describes the interface of the furniture-sale service. This service allows the controller to request information about a particular product (*prodInfoReq*) and replies with the required information (*infoRcvd*). Then, it allows placing (*prodReq*) or canceling (*prodCancel*) the request. The delivery-booking service is described in Fig. 1b. It allows the controller to request shipping information (*shipInfoReq*) and replies with the shipping time and cost (*costAndTime*). Then, the controller can cancel the order (*shipCancel*) or require the shipping of the product (*shipReq*). A model describing the user interaction with the composed web-service is shown in Fig. 1c. The user requests information about some product (*usrReq*) and receives an offer including both shipping and product information (*offerRcvd*). Whenever an offer is received, the user can confirm (*userAck*) or decline it (*userNack*). If the order is confirmed, the user waits for the shipment (*respOk*).

The controller needs to ensure that the system satisfies the following properties: (P1) the p&d system must only check

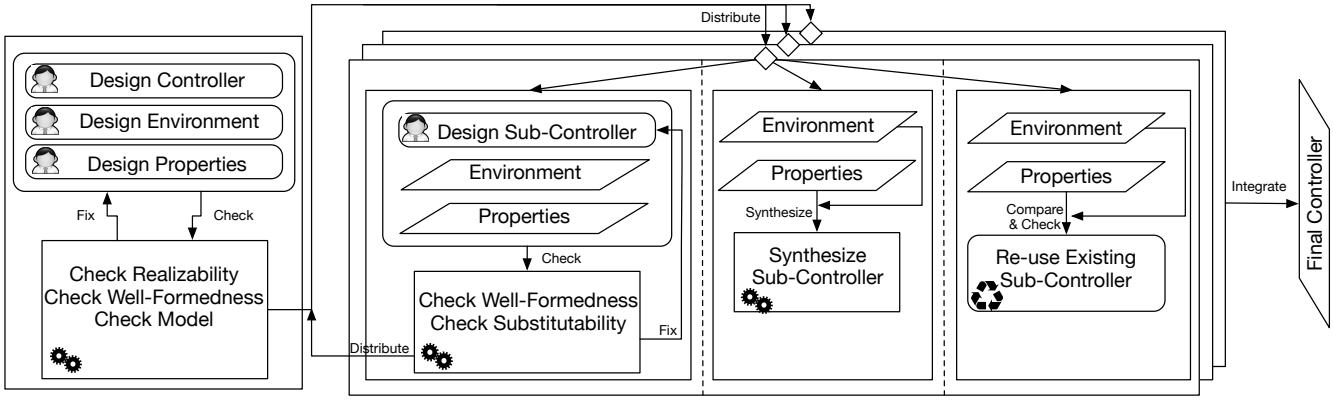


Fig. 2: Overview of FIDDLE.

for some product or shipping info if the user has placed an order; (P2) an offer is provided to the user only if the furniture and shipping services have confirmed the availability of the requested product; (P3) furniture and shipping are activated only if the user has decided to purchase; (P4) the p&d system allows canceling an order only if the cancellation procedure was initiated by a user; (P5) a request is marked as canceled when both the product ordering and the shipping services have canceled it; (P6) the p&d service finishes successfully only when both the product ordering and the shipping service have handled their requests correctly; (P7) the controller guarantees that infinitely many products can be shipped.

III. Overview

In this section, we describe FIDDLE – our verification-driven environment supporting incremental and distributed controller development. A high-level view of our approach is presented in Fig. 2. The approach combines steps in which the human insight and experience are exploited (the rounded boxes labeled with a designer icon or a recycle symbol, to indicate design or reuse, respectively) and those where automated support is provided for verifying the current state of the design, synthesizing parts of the controller, or checking whether the designed sub-controller can correctly fit into the original design (squared boxes labeled with a pair of gearwheels).

Creating the initial controller design. A design team models the environment to be controlled, formalizes the properties the controller has to guarantee and designs an initial, high-level structure of the controller. The environment interface is formalized using finite state machines in which each transition is labeled with an event that can be spontaneously generated by the environment or triggered by the controller. The environment can comprise different components and its overall behavior is obtained by combining their individual behaviors. An environment of the p&d system can be obtained by combining the components in Figs. 1a, 1b, and 1c.

Designers also formulate properties the controller needs to ensure. In FIDDLE they are expressed in Fluent Linear Temporal Logic (FLTL) [12], [13], [14], described in Sec. IV.

The controller is modeled using a state-based formalism that can clearly identify parts (called "sub-controllers" in this paper), expressed as *black boxes*, whose internal design is delayed to a later stage or set apart for distributed development by other parties. Black boxes are enriched with an interface that provides information on how the sub-controllers are to be connected with the rest of the controller and which kinds of "actions" each of them can perform The black boxes are further decorated with pre- and post-conditions, allowing distributed teams to develop sub-controllers without the need to know about the rest of the system. Pre- and post-conditions are formalized using FLTL.

A possible initial design for the controller in the p&d example is shown in Fig. 1e. This represents a simple architecture with an initial fully specified state 1 and a black box state 2. In the model, whenever a *userReq* event is detected, the controller moves into the black box state 2, which represents a sub-controller in charge of managing the user request. As the sub-controller is exited, it generates the events *respOk* or *reqCanc* depending on whether the request is confirmed or canceled. To specify state 2, we add a pre-condition that there is a user request that has not yet been handled, and a post-condition that the request has either been sent both to the shipping and the furniture-sale services, or it has been cancelled.

Analyzing partial controllers. Once these initial artifacts are created, they are analyzed through a set of automated controller architecture checkers: *realizability*, *well-formedness*, and *model checking*. The *realizability checker* confirms the existence of a system which can control the given environment to guarantee the properties of interest. If such a system does not exist, no further development is needed. Rather, the designer needs to rethink the properties and/or the environment model. The *well-formedness checker* verifies that both the pre- and the post-conditions on black boxes are satisfiable. Finally, the *model checker* verifies whether the (partial) controller guarantees satisfaction of the properties of interest.

In our example, the model checker identifies a problem with the structure of the controller: regardless of how state 2 will be defined, the controller cannot satisfy properties P1 and P7. This suggests that the designer needs to change the

existing pre- and post- conditions and/or the current controller model to be able to guarantee these properties. For example, the development team might want to replace the initial design with a more complex one, like the one shown in Fig. 1d. This new model includes two regular states: state 1, in which the controller waits for a new user request, and state 3, in which the controller has provided the user with an offer and is waiting for an answer. The user might accept (*userAck*) or reject (*userNack*) an offer and depending on this choice, either state 4 or 5 is entered. States 2, 4 and 5 are black boxes, to be refined later. The developers also need to provide the specification for the unspecified states. In our example, the pre- and post-conditions of box 2 specify that there is a pending user request, and that cost, time and product information are collected. Pre-condition of box 4 guarantees that *offerRcvd* has occurred after the user request, and its post-condition assures that both a product and shipping requests are performed. Finally, pre-condition of box 5 assures that *offerRcvd* has occurred after the user request and before entering the state and its post-condition ensures that both the product and the shipping requests are canceled when the state is exited. This model is checked using the provided tools, and, since it passes all the checks, it is used in the next phase of the development.

Architecture refinement and sub-controller development. Once the design satisfies the properties of interest, the design team may choose to refine the controller structure (and keep improving it until they are satisfied with the result), or *distribute* the development of unspecified sub-controllers to other (internal or external) development teams, including their own.

Each team can then develop the assigned sub-controller using any available technique, including reusing existing sub-controllers or synthesizing new ones from the provided specifications. The only constraints are (1) given the stated pre condition, the obtained sub-controller has to satisfy its post-condition, and (2) the sub-controller should operate in the same environment as the overall controller.

Sub-controller development can itself be an iterative process, and FIDDLE supports its development with a process similar to the one we have just described (i.e., with manual development of a partial sub-controller, well-formedness checking, etc.), but neither the model of the environment nor the overall properties of the system can be changed during this process. Otherwise, the resulting sub-controller cannot be automatically integrated into the overall system.

In our example, we can delegate the development of the sub-controller for state 2 to an external contractor who can choose any technique, including controller synthesis or reuse of an existing sub-controller, as long as it satisfies the given post-condition. Possible candidate replacements for the black box are the sub-controller in Fig. 1f or the one in Fig. 1g. In the former case, the controller asks for shipping info details and waits until the shipping service provides cost and time. Then it queries the furniture-sale service to obtain the product info. In the latter, the shipping and the furniture services are queried and then the controller waits for the corresponding answers. Since these candidates are not partially defined, the well-formedness checking is not needed. Yet the substitutability checking is still needed to ensure that the candidate replacement satisfies the post-condition, given the pre-condition. The pre- and post-conditions defined in the previous section for state 2 pass this check.

Sub-controller integration. The formalism and the analyses supported by FIDDLE guarantee that if each sub-controller is developed correctly w.r.t the pre- and post-conditions of the corresponding black box, the resulting controller is also correct. In our example, since both of the proposed candidates (Figs. 1f and 1g) pass the substitutability check, either can be used instead of state 2 in Fig. 1d, and the resulting controller would still guarantee the properties of interest.

IV. Preliminaries

The model of the *environment* E specifies the main characteristics of the domain in which the controller is executed. It describes how the environment influences the controller, and how it reacts to controller actions. The *controller* C modifies the behavior of the environment to guarantee the satisfaction of the properties of interest. The behavior of the final *system* S is obtained by parallel composition between E and C. C is *correct* if S satisfies the desired properties.

Models and operations. Controllers and environments are modeled using *Labelled Transition Systems* [15], [7]. Let Act be the universal set of observable actions and let $Act_\tau = Act \cup \{\tau\}$, where τ denotes an unobservable local action.

Definition 1: (Labeled Transition System (LTS) [16]) An *LTS* is a tuple $A = \langle Q, q_0, \alpha A, \Delta \rangle$, where Q is the set of states, $q_0 \in Q$ is the initial state, $\alpha A \subseteq Act$ is a finite set of actions, and $\Delta \subseteq Q \times \alpha A \cup \{\tau\} \times Q$ is the transition relation.

Given an LTS $A = \langle Q, q_0, \alpha A, \Delta \rangle$, $\pi = l_0, l_1 \ldots$ is an *infinite trace* of A if there exists a sequence $q_0, l_0, q_1, l_1, \ldots$, where for every $i \geq 0$, $(q_i, l_i, q_{i+1}) \in \Delta$.

Given a state q, let Δ_q^- and Δ_q^+ denote the sets of its incoming and outgoing transitions, respectively, and Δ_q denote the union of the incoming and outgoing transitions of q. Given a transition $\delta = (q_1, l, q_2)$, we use δ^- (δ^+) to indicate the source (destination) q_1 (q_2) of δ, and δ_l for its label l. For example, transition δ = (2,*infoRcvd*,3) of the LTS of the environment in the p&d example described in Fig. 1a connects states 2 and 3 and is labeled with the event *infoRcvd*. Here $\delta^- = 2$, $\delta^+ = 3$ and δ_l =*infoRcvd*.

Definition 2: (Parallel Composition [11]) Let $M = \langle Q_M, q_M^0, \alpha M, \Delta_M \rangle$ and $N = \langle Q_N, q_N^0, \alpha N, \Delta_N \rangle$ be LTSs. *Parallel composition* $\parallel$ is a symmetric operator such that $M \parallel N$ is the LTS $P = \langle Q_M \times Q_N, q_M^0 \times q_N^0, \alpha M \cup \alpha N, \Delta \rangle$, where Δ is the smallest relation that satisfies the following rules, where $l \in \alpha M \cup \alpha N \cup \{\tau\}$:

- $\frac{(s,l,s') \in \Delta_M}{(\langle s,t \rangle, l, \langle s',t \rangle) \in \Delta}, l \in \alpha M \setminus \alpha N \text{ or } l = \tau$
- $\frac{(t,l,t') \in \Delta_N}{(\langle s,t \rangle, l, \langle s,t' \rangle) \in \Delta}, l \in \alpha N \setminus \alpha M \text{ or } l = \tau$;
- $\frac{(s,l,s') \in \Delta_M, (t,l,t') \in \Delta_N}{(\langle s,t \rangle, l, \langle s',t' \rangle) \in \Delta}, l \in \alpha N \cap \alpha M, l \neq \tau$.

The rules indicate that there is synchronization on shared events and interleaving on non-shared ones.

Properties. The properties of interest are often specified using Fluent Linear Time Temporal Logic (FLTL) [12], [13], [14]. A *fluent* is a property of the world that holds after it is initiated by an action and ceases when it is terminated by another action.

Definition 3: (Fluent [12]) A *fluent* Fl is a pair $\langle I_{Fl}, T_{Fl}\rangle$, where $I_{Fl} \subset Act$, $T_{Fl} \subset Act$ and $I_{Fl} \cap T_{Fl} = \emptyset$.

A fluent may be *true* or *false*. A fluent is *true* if it has been initialized by an action $i \in I_{Fl}$ at an earlier time point and has not yet been terminated by another action $t \in T_{Fl}$; otherwise, it is *false*. The initial value of the fluent is specified using the attribute $Initially_{Fl}$ [17].

Given a set of fluents Φ and an infinite trace $\pi = l_0, l_1, \ldots$ over Act, *an FLTL interpretation* of π is an infinite trace $f_0, f_1, \ldots$ over 2^{Φ} which assigns to each index i of π the set of fluents that hold in position i. Formally, $\forall i \in N, \forall Fl \in \Phi$, $Fl \in f_i$ both of the following conditions hold:

- $Initially_{Fl} \wedge (\forall k \in N, 0 \leq k \leq i, l_k \notin T_{Fl})$;
- $\exists j \in N \mid ((0 \leq j \leq i) \wedge (l_j \in I_{Fl}) \wedge (\forall k \in N, j \leq k \leq i, l_k \notin T_{Fl}))$.

For example, consider the LTS in Figure 1c and the fluent *F_ReqPend*=$\langle\{$*userReq*$\}, \{$*respOk, reqCanc*$\}\rangle$, which is initially *false*. *F_ReqPend* holds in a trace of the LTS from the moment at which *userReq* occurs and until a transition labeled with *respOk* or *reqCanc* is fired.

An FLTL formula is obtained by composing fluents with standard LTL operators: $\bigcirc$ (next), $\Diamond$ (eventually), $\Box$ (always) and $\mathcal{U}$ (until). For example, the property P1 of the p&d example is expressed in FLTL as $\Box((F\_ShipInfoReq \vee F\_ProdInfoReq) \rightarrow F\_UsrReq)$, where *F_ShipInfoReq* and *F_ProdInfoReq* are appropriate fluents.

FLTL formulae can also be interpreted on finite traces by connecting fluents with temporal logic operators of the "finite LTL" language LTL_f [18]. When such an interpretation is desired, we denote the resulting logic by $FLTL_f$.

Verification. The verification procedure [14] checks whether the system S, obtained by combining the environment and the controller, satisfies a given FLTL formula ϕ by translating it into an automaton which is synchronously executed with S. The resulting automaton is checked for emptiness.

V. Modeling and Refining the Controller

This section introduces a novel formalism for modeling and iteratively refining controllers.

A. Interface Partial LTS

We define the notion of partial LTS and then extend it with interface specification. Intuitively, a partial LTS is an LTS where some states are "regular" and others are "black box". Black box states model portions of the controller whose behavior still has to be specified. Black box states are augmented with interface specification: the set of events that can be monitored and controlled by the unspecified boxes as well as pre- and post-conditions on their behavior.

Partial LTS. The set of states of the LTS is partitioned into *regular* and *black box* states, and each black box is associated with an *interface*.

Definition 4: (Partial LTS) A *Partial LTS (PLTS)* is structure $P = \langle A, R, B, \sigma\rangle$ where:

- $A = \langle Q, q_0, \alpha A, \Delta\rangle$ is an LTS;
- Q is the set of states, s.t. $Q = R \cup B$ and $R \cap B = \emptyset$;
- R is the set of *regular* states;
- B is the set of *black box* states;
- $\sigma : B \rightarrow 2^{\alpha A}$ is the *interface*.

An LTS is a PLTS where the set of black box states is empty.

For example, the PLTS in Fig. 1d is defined over the regular states 1 and 3 and the black box states 2, 4 and 5. The function σ specifies that the box 2 monitors the occurrence of *infoRcvd* and *costAndTime* and acts over *prodInfoReq* and *shipInfoReq*.

Let P be a PLTS $\langle A, R, B, \sigma\rangle$ defined over the LTS $A = \langle Q, q_0, \alpha A, \Delta\rangle$. An infinite sequence $\pi = l_0, l_1 \ldots$ is a *required trace* if there exists a sequence $q_0, l_0, q_1, l_1, \ldots$ where for every $i \geq 0$ we have $(q_i, l_i, q_{i+1}) \in \Delta$, $q_i \in R$ and $q_{i+1} \in R$. An infinite sequence $\pi = l_0, l_1 \ldots$ is a *maybe trace* if it is not a required trace and there exists a sequence $q_0, l_0, q_1, l_1, \ldots$ where for every $i \geq 0$ we have that either $(q_i, l_i, q_{i+1}) \in \Delta$, $q_i \in R$ and $q_{i+1} \in R$, or $q_i = q_{i+1}$, $q_i \in B$ and $l_i \in \sigma(q_i)$. A sequence $\pi = l_0, l_1 \ldots$ is a *possible trace* of P if it is a required or a maybe trace. For example, the PLTS of Fig. 1d only contains maybe traces, since each trace goes through at least one black box state.

Definition 5: (Parallel execution between PLTS and LTS) Given a PLTS $P = \langle A, R, B, \sigma\rangle$ defined over the LTS $A = \langle Q^A, q_0^A, \alpha A, \Delta^A\rangle$ and an LTS $B = \langle Q^B, q_0^B, \alpha B, \Delta^B\rangle$ the parallel execution is an LTS $S = \langle Q^S, q_0^S, \alpha S, \Delta^S\rangle$ such that $Q^S = Q^A \times Q^B$; $q_0^S = (q_0^A, q_0^B)$; $\alpha S = \alpha B \cap \alpha A$; and the set of transitions Δ^S is defined as follows:

- $\frac{(s,l,s') \in \Delta_B}{(\langle s,t\rangle, l, \langle s',t\rangle) \in \Delta_S}, l \in \alpha B \setminus \alpha A \text{ or } l = \tau \text{ or } t \in B$
- $\frac{(t,l,t') \in \Delta_A}{(\langle s,t\rangle, l, \langle s,t'\rangle) \in \Delta_S}, l \in \alpha A \setminus \alpha B \text{ or } l = \tau$;
- $\frac{(s,l,s') \in \Delta_A, (t,l,t') \in \Delta_B}{(\langle s,t\rangle, l, \langle s',t'\rangle) \in \Delta_S}, l \in \alpha A \cap \alpha B, l \neq \tau$.

Given P, A, B defined above, the system $S = P \parallel B$ and a state s of P, we say that a finite trace $l_0, l_1, \ldots l_n$ of S *reaches* s if there exists a sequence $\langle q_0^A, q_0^B\rangle, l_0, \langle q_1^A, q_1^B\rangle, \ldots l_n, \langle s, q_{n+1}^B\rangle$, where for every $0 \leq i \leq n$, we have $(\langle q_i^A, q_i^B\rangle, l_i, \langle q_{i+1}^A, q_{i+1}^B\rangle) \in S$. For example, considering the LTS in Fig. 1c and the PLTS in Fig. 1d, the finite trace obtained by performing a *userReq* event reaches the black box state 2 of the PLTS.

Given a finite trace $\pi = l_0, l_1, \ldots l_n$ (or an infinite trace $l_0, l_1, \ldots$) of S, we say that its sub-trace $l_i, l_{i+1} \ldots l_k$ is *inside* the black box b if one of the sub-sequences associated with π is in the form $\langle b, q_i^B\rangle, l_i, \langle b, q_{i+1}^B\rangle, \ldots, l_n, \langle b, q_k^B\rangle$, where $l_i, l_{i+1}, \ldots, l_n \in \sigma(b)$. For example, considering the parallel execution of the LTSs in Figs. 1c and 1b and the PLTS in Fig. 1d and the finite trace *userReq, shipInfoReq, offerRcvd*, the sub-trace *shipInfoReq* is inside the black box 2.

Adding pre- and post-conditions. The intended behavior of sub-controllers can be captured in the form of pre- and post-conditions.

Definition 6: (IPLTS) An *interface PLTS* I is a PLTS structure $P = \langle A, R, B, \sigma \rangle$ where every $b \in B$ is annotated with a pair $\langle \varepsilon, \gamma \rangle$ s.t. ε and γ are FLTL$_f$ formulae.

ε is *a pre-condition* that specifies a constraint that must be satisfied by all finite traces of P that reach b. For example, consider the FLTL$_f$ pre-condition $\Diamond\Box$*F_CostAndTime* (defined over the fluent *F_CostAndTime*) for the black box 4 of the IPLTS in Fig. 1d. This pre-condition ensures that any trace of the composition between the IPLTS and a LTS needs to receive the cost and time info before reaching this state.

The FLTL$_f$ formula γ is *a post-condition* that constrains the behavior of the system in any sub-trace performed inside box b. For example, consider the post-condition $\Diamond\Box$*F_ProdReq* of the black box 4 of the IPLTS in Fig. 1d. It ensures that whenever this IPLTS is composed with an LTS, a product request needs to be performed by the furniture-sale service before exiting the black box.

Given an IPLTS I and an LTS B, the *parallel execution* S between I and B is obtained by considering the PLTS P associated with I and the LTS B as specified in Def. 5. Given a trace of S, if the post-condition holds in any sub-trace performed inside box b, the trace is called *valid*.

Definition 7: (Well-formed IPLTS) Given an LTS B, an IPLTS I is *well-formed* (over B) iff every valid trace of $S = I \parallel B$ satisfies all the pre-conditions of I's black box states.

We say that $S = I \parallel B$ *satisfies* the FLTL property ϕ, if and only if ϕ is satisfied by every valid trace of S. In the p&d example, the post-condition $\Diamond$(*F_ProdReq_Post*) $\wedge$ $\Diamond$(*F_ShipReq_Post*) is shown to guarantee that the system satisfies P3. The full formalization of the motivating example can be found at goo.gl/531rXD.

B. IPLTS refinement

We now define the process of refining IPLTSs. Intuitively, refinement aims to replace black-box states of a given IPLTS with sub-controllers that satisfy their pre- and post-conditions.

Definition 8: (Sub-controller) Given a black-box b and its interface αb, a *sub-controller for the box* b is an IPLTS R defined over the set of events αb. One state q_f of R is defined as the *final state* of R.

Given a sub-controller R and the environment E, and a trace in the form $\pi_i\pi_e$ such that $\pi_i = l_0, l_1 \dots l_n$ and $\pi_e = l_n, l_{n+1}, \dots l_k$, we say that $\pi_i\pi_e$ is *a trace of the parallel execution between* R *and* E iff 1) there exists a sequence $q_0, l_0, q_1, l_1 \dots l_n, q_n, l_{n+1}$ in the environment such that (q_i, l_i, q_{i+1}) is a transition of E; 2) π_e is obtained by $R \parallel E$, considering l_{n+1} as initial state for the environment, 3) π_e reaches q_f. A sub-controller is *valid* if it ensures that the traces of the parallel execution satisfy its post-conditions. Intuitively, a trace of the parallel execution between a sub-controller R and the environment E is obtained by concatenating two sub-traces: π_i and π_e. The sub-trace π_i corresponds to a set of transitions performed by the environment before the the sub-controller is activated, while π_e is a trace the system generates while it is in the sub-controller R.

Definition 9: (Substitutable sub-controller) Let R be a sub-controller with a pre-condition ϵ and a post-condition γ, and E be a model of the environment. R is a *substitutable sub-controller* iff for every trace of $R \parallel E$ computed from the state s of E, if π_i is a finite path which reaches s and satisfies ϵ then π_e satisfies γ.

Intuitively, whenever the sub-controller is entered and the pre-condition ϵ is satisfied (i.e., the trace π_i satisfies ϵ), then a trace of the parallel execution between the sub-controller and the environment that reaches the final state of the sub-controller must satisfy the post-condition.

Sub-controller integration. A black-box of an IPLTS C can be replaced by a substitutable sub-controller R resulting in an IPLTS C' which refines C.

Definition 10: (Integration) Given a well formed IPLTS $C = \langle A, R, B, \sigma \rangle$ with $A = \langle Q, q_0, \alpha A, \Delta \rangle$, and a substitutable sub-controller $R = \langle A^R, R^R, B^R, \sigma^R \rangle$ for a box $b \in B$ with final state q_f and $A^R = \langle Q^R, q_0^R, \alpha A^R, \Delta^R \rangle$, the refinement C' is an IPLTS $\langle A', R', B', \sigma' \rangle$ such that:

- A' is a LTS $\langle Q', q_0', \alpha A', \Delta' \rangle$, where
  - $Q' = Q \cup Q^R \setminus \{b\}$;
  - $q_0' = \begin{cases} q_0 & \texttt{if } q_0 = b; \\ q_0^R & \texttt{if } q_0 \neq b; \end{cases}$
  - $\alpha A' = \alpha A^R \cup \alpha A$;
  - $\Delta' = \Delta \cup \Delta^R \setminus \Delta_b \cup \{(\delta^-, \delta_l, q_0') \mid \delta \in \Delta_b^-\} \cup \{(q_f', \delta_l, \delta^+) \mid \delta \in \Delta_b^+\}$.
- $R' = R \cup R^R$;
- $B' = B \cup B^R \setminus \{b\}$;
- $\sigma'(b) = \begin{cases} \sigma(b) & \texttt{if } b \in B \setminus \{b\}; \\ \sigma^R(b) & \texttt{if } b \in B^R \end{cases}$

For example, integrating the sub-controller R in Fig. 1f into the refined controller in Fig. 1d produces the IPLTS in Fig. 3. The prefix "2." is used to identify the states obtained from R.

Theorem 1: (Correctness of integration) Integrating a substitutable sub-controller R into a well-formed IPLTS C that satisfies a property ϕ preserves satisfaction of ϕ.

The proof of this and other theorems can be found in the extended version of the paper (http://goo.gl/SUXATG).

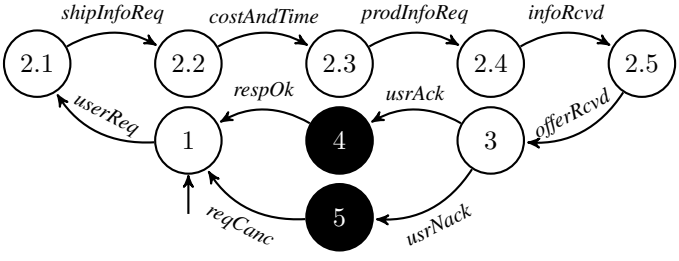


Fig. 3: Controller obtained by plugging in the sub-controller from Fig. 1f into the IPLTS from Fig. 1d.

Proof: Since C is well formed, by Def. 7, every valid trace of the integration C' satisfies the pre-conditions of the boxes. Since R is substitutable, by Def. 9, if the pre-conditions are satisfied then so are the post-conditions; thus, the system will exhibit only valid traces that satisfy the post-conditions. So, all traces of C' are valid and satisfy the pre-condition of the boxes. Since the set of the valid traces of C' is a sub-set of the valid traces of C, and every valid trace of C satisfies ϕ, C' also satisfies ϕ. ■

In our example, if the sub-controller R from Fig. 1f is substitutable, integrating it into the IPLTS C of Fig. 1d ensures that the integrated controller C' preserves the satisfaction of properties P1-P6.

VI. Verification algorithms and implementation

In this section, we describe the algorithms for the analysis of partial controllers and their implementation.

Checking realizability. Realizability of a property ϕ is checked via the following procedure. Let C^B be the LTS resulting from removing all black boxes and their incoming and outgoing transitions from the given partial controller C. Check $E \parallel C^B \models \phi$. If ϕ is not satisfied, the controller is not realizable. Otherwise, compute $E \parallel C$ (as specified in Def. 5) and model-check it against $\neg\phi$. If the property is satisfied, the controller is not realizable: no matter how the boxes will be refined, all the behaviors of the system will not satisfy ϕ. Otherwise, the controller is realizable. Thus, realizability checking reduces to two classical FLTL model checking runs.

Theorem 2: (Correctness) The realizability checker returns "realizable" iff there exists a controller C', obtained from C by integrating sub-controllers, s.t. $E \parallel C' \models \phi$.

Proof: If $E \parallel C^B \not\models \phi$, there exists a trace of the environment which is permitted by the already specified portion of the controller that violates ϕ. Thus, the controller is not realizable.

If $E \parallel C \not\models \neg\phi$, there exist some traces π of E on which ϕ holds. Thus, a controller that constrains the environment to expose only π traces can exist. ■

Checking well-formedness. Given a partial controller C with a black-box b annotated with a pre-condition ϵ, well-formedness checks whether ϵ is satisfied in C as follows:

1. Transform post-conditions into LTSs. Translate every FLTL_f post-condition γ of every black box state of the controller, including b, into an FLTL formula γ' as specified in [19]. This transformation ensures that the *infinite* traces that satisfy γ' have the form $\pi, \{end\}^\omega$, where π satisfies γ. For each box b_i, the corresponding post-condition γ' is transformed into an equivalent LTS, named LTS_{b_i}, using the procedure in [3]. Since the LTS_{b_i} has traces in the form $\pi, \{end\}^\omega$, it has a state s with an *end*-labelled self-loop. This self-loop is removed and s is considered as final state of LTS_{b_i}. All other *end*-labeled transitions are replaced by τ transitions. The computed automata LTS_{b_i} contain all the traces that do not violate the corresponding post-conditions and their size is, in the worst case, exponential in the size of the corresponding post-condition. This step is performed since the pre-condition of the black box b must be verified under the assumption that all black boxes, including b itself, satisfy their post-conditions.

2. Integrate the LTSs of all the boxes $b_i \neq b$. For every black box state $b_i \neq b$, eliminate b_i and add the LTS_{b_i} (constructed using procedure in step 1) to C by replacing every incoming transition of b_i with a transition whose destination is the initial state of LTS_{b_i} and every outgoing transition of b_i with a transition whose source is the final state of LTS_{b_i}. This step creates an LTS which encodes all the traces of the controller that do not violate any post-conditions of its black boxes.

3. Integrate LTS of the black box b into the controller. Integrate LTS_b into C together with two additional states, q_1 and q_2, calling the resulting model C'. Specifically, replace every incoming transition of b by a transition with destination q_1. Replace every outgoing transition of b by a transition whose source is the final state of LTS_b. Add a transition labeled with τ from q_1 to the initial state of LTS_b. Add a self-loop labeled with en event *end* (which is not in the set of events of the enironment or the controller) to q_2. Add a τ transition from q_1 to q_2. For every event of the environment E that does not belong to the controller C, add a self-loop to every state of the controller with the exception of state q_2. The obtained LTS encodes all the valid traces of the system. When a valid trace reaches the black box b, C' can enter state q_2, from where only the *end*-labelled self-loop is available.

4. Verify. Translate the FLTL_f formula ϵ into an FLTL formula, called ϵ', as specified in [19]. The above transformations ensure that if a trace in the form $\pi, \{end\}^\omega$ satisfies ϵ' then π satisfies ϵ, and $C' \parallel E$ has a valid trace of this form that reaches the black box b. To check the precondition, run the model-checker to verify whether $C' \parallel E \models \epsilon'$.

Thus, well-formedness checking reduces to a classical FLTL model checking performed over $C' \parallel E$ and the FLTL formula ϵ'. The size of ϵ' is proportional to the size of ϵ. C' is obtained by plugging the LTSs associated with the post-conditions of boxes into the original controller.

Theorem 3: (Correctness) The well-formedness procedure returns *true* iff all of the valid traces of C satisfy the pre-condition ϵ.

Proof: ($\Rightarrow$) The proof is by contradiction. If there exists a valid trace that does not satisfy ϵ, it must be a finite trace π of $C \parallel E$ that reaches b. By construction, whenever the black box b is entered, the LTS C' can execute a τ transition reaching a state with an *end*-labeled self-loop. Thus, a trace in the form $\pi, \tau, \{end\}^\omega$ is in $C' \parallel E$. Since π does not satisfy ϵ, the trace $\pi, \tau, \{end\}^\omega$ violates ϵ'. Thus, well-formedness procedure returns *false* which is a contradiction.

($\Leftarrow$) If all of the valid traces of C satisfy ϵ, There is no finite trace π of $C' \parallel E$ that reaches b and violates ϵ. Thus, by construction, there cannot exist a trace of the form $\pi, \tau, \{end\}^\omega$ in $C \parallel E$ where π violates ϵ. Thus, checking ϵ' in $C' \parallel E$ returns *true*. ■

Model checking. To check whether $E \parallel C$ satisfies ϕ, we first construct an LTS C' that generates only valid traces, by

plugging into C the LTSs corresponding to all its black boxes (as done in steps 1 and 2 of the well-formedness checking) and use a classical FLTL model-checker to verify $E \parallel C' \models \phi$.

Theorem 4: The model checking procedure returns *true* iff every valid trace of $E \parallel C$ satisfies ϕ.

Proof: ($\Rightarrow$) The proof is by contradiction. If there exists a valid trace of $E \parallel C$ that does not satisfy ϕ, then, it must also be a trace of $E \parallel C'$ since the traces of $E \parallel C'$ are the valid traces of $E \parallel C$. Thus, the model checking procedure returns *false*.

($\Leftarrow$) By construction, the traces of $E \parallel C'$ are the valid traces of $E \parallel C$. ∎

Checking substitutability. Given a controller C with a black-box b annotated with $\langle \epsilon, \gamma \rangle$ and a sub-controller R, this procedure checks whether R can substitute b into C. We first discuss the case in which the sub-controller R does not contain black box states. The procedure goes as follows:

1. Transform the pre-condition ϵ into an LTS, called LTS(A_ϵ), using step 1 of the well-formedness procedure.

2. Compute the sequential composition ($A_\epsilon.R$) between A_ϵ and R. This is done by connecting the final state s of A_ϵ with the initial state of R by a transition labelled with a fresh event *init*. The final state of R if then connected to an additional state s through a τ-labeled transition. A self-loop labeled with a fresh event *end* is added to s. By performing these steps, the prefix π of every infinite trace in the form $\pi\{end\}^\omega$ is made by two parts: π_1, which satisfies the pre-condition, and π_2, which is generated by the LTS R.

3. Verify the result. Translate the FLTL$_f$ formula $\lambda = init \rightarrow \bigcirc\gamma$ into an FLTL formula λ' as in [19]. A trace in the form $\pi, \{end\}^\omega$ satisfies λ' if and only if π satisfies λ. Check $E \parallel (A_\epsilon.R) \models \lambda'$ with a classical model-checker.

Theorem 5: The substitutability checker returns *true* iff for every trace π_i, π_e of $E \parallel R$ computed starting from the state s of the environment, if π_i is a finite trace that reaches s and satisfies ϵ then π_e satisfies γ.

Proof: ($\Rightarrow$) If the substitutability checker returns *true*, then every trace of $E \parallel (A_\epsilon.R)$ satisfies the property $\lambda' = init \rightarrow \bigcirc\gamma$. By construction the traces of $E \parallel (A_\epsilon.R)$ are of the form $\pi_i, init, \pi_e$. The sub-trace π_i is a finite trace of the environment E that satisfies the pre-condition and reaches s, by construction. Since $\pi_i, init, \pi_e$ satisfies λ', the sub-trace π_e satisfies γ on a finite trace of $E \parallel R$ that reaches a final state of R.

($\Leftarrow$) The proof is by contradiction. If the substitutability checker returns *false*, then there exists a trace $\pi_i, init, \pi_e$ of $E \parallel (A_\epsilon.R)$ that does not satisfy λ'. Since the property is not satisfied, π_e violates γ. By construction, π_i, π_e is also a trace of $E \parallel R$ and π_i satisfies ϵ. Thus, there exists a finite trace π_i, π_e of $E \parallel R$, where s is the final state of π_i, s.t. π_i satisfies ϵ and π_e violates γ. ∎

Note that, if the sub-controller R contains black boxes, R is modified by performing steps 1 and 2 of the well-formedness checking before running the substitutability procedure.

Tool support. We implemented our algorithms on top of LTSA [20]. Our implementation is available at goo.gl/531rXD.

VII. Evaluation

This section reports on our experience evaluating the FIDDLE and aims to answer two questions: **RQ.1**: *How effective is the proposed approach w.r.t supporting iterative, distributed development of correct controllers?* and **RQ.2**: *How scalable is the automated part of the proposed approach?*

A. Assessing effectiveness

Methods and models. Rather than simulating forward development of a complex controller, we chose to perform a "redesign" exercise by starting with an existing controller, abstracting it to produce a partial system, annotating black boxes with pre- and post-conditions, and using documentation that came with the controller to understand its desired properties and improve the controller. We also simulated distributed development by using specifications of black boxes to construct sub-controllers.

We chose the *Executive* module of the K9 Mars Rover developed at NASA Ames [21], [22], [4] and specified using LTSs[1]. The *Executive* is a multi-threaded system with the following components: *Executive* – the controller of the system; *ExecCondChecker* – a component for monitoring the state conditions, and *ActionExecution* – a component for sending commands to the Rover. *ExecCondChecker* is further decomposed into *db-monitor* and *internal*. The synchronization between these modules is performed through shared variables (with locks). The overall size of the *Executive* is ~10^6 states.

To simulate forward development of partial controllers, we encapsulated a portion of *Executive* into a black box state and took the other components as its environment. Specifically, the original *Executive* controller contained states WaitingForPlan (*S1*), PrepareExecution (*S2*), ExecutePlan (*S3*), ExecuteCurrentNode (*S4*), ExecuteTaskAction (*S5*), WaitForTermination (*S6*) and ClearConditions (*S7*). In the original model, each of these states was further decomposed into an LTS. The Mars Rover controller was obtained by replacing these states with the corresponding LTSs. At the end, it had about 100 states. To generate a partial controller, we added the black box state PlanManager (*S8*) in order to replace $S3$-$S6$.

Properties to be satisfied by the resulting Mars Rover controller have been inspired by comments contained in the text version of its model: the *Executive* forces an action to be executed only after the controller gets the *exec* lock (*P1*); the lock over the *condList* variable is obtained only after obtaining the *exec* lock (*P2*); the *Executive* forces an action to be executed only after reading a plan (*P3*); the *Executive* ensures that a plan is not read during the execution of an action (*P4*); *Executive* ensures that a lock on *condList* is obtained before accessing the variables of *execCondChecker* (*P5*).

Constructing the initial controller model. As a first step, we verified whether the partial model of Executive satisfies the properties of interest using the *model checker*. Interestingly,

[1] All the evaluation results are available at goo.gl/531rXD.

P3 was already satisfied by the partial design. Indeed, before entering the PlanManager, the plan is read while *Executive* is in the state WaitingForPlan. Then we used the *realizability checker* to determine whether it is possible to guarantee satisfaction of the other properties which was affirmative. Afterwards, we attempted to develop and check pre- and post-conditions of PlanManager. For example, by using *well-formedness checker*, we checked whether every trace entering PlanManager satisfied its pre-conditions, e.g., whether PlanManager had the lock on the *exec* whenever the box was entered. Surprisingly, this property was not satisfied. Indeed, there was no post-condition that forced the black-box to not release the lock over *exec*. Thus, the second time the black box was entered, *Executive* did not have the lock. Forcing *Executive* to have the lock on *exec* upon exiting PlanManager through an appropriate post-condition was sufficient to guarantee the satisfaction of the pre-condition.

We then defined an additional set of post-conditions to ensure the satisfaction of the properties of interest. For example, to ensure the satisfaction of property *P1*, we required that whenever an action is executed, PlanManager ensures that the controller has the lock for *exec*. Surprisingly, this condition was not sufficient, as we discovered by running the *model checker*. Indeed, the PlanManager component could be exited while the action was still being executed. The rest of the model would then release the *exec* lock (without terminating the action), thus violating the property. This feedback allowed us to strengthen the post-condition of PlanManager to require that upon exiting the component, the action must be terminated. To summarize, we were able to use the support provided by FIDDLE to define and iteratively refine pre- and post-conditions of PlanManager. We needed to run the well-formedness analyzer and the model checker and iteratively inspect the returned violating traces to understand which post-conditions were not ensuring the properties of interest or satisfying pre-conditions of the black-box.

Distributed sub-controller development. Given a single black-box, we could not simulate distributing the work for designing sub-controllers to multiple teams. But we did simulate the process of sub-controller development by using only the pre- and post-conditions of PlanManager rather than the entire controller.

We considered two sub-controllers, $C1$ and $C2$, as replacements for PlanManager. $C1$ is the portion of the original *Executive* controller that we abstracted to produce PlanManager. $C2$ is a variation of the sub-controller in which the lock over the *condList* variable is removed. $C2$ is a viable alternative because, by inspecting the comments in the Mars Rover model, we realized that the developer was uncertain whether to put a lock on the *condList* before performing an assignment over a variable of Executive.

We used our *substitutability* checker to determine suitability of $C1$ and $C2$ as implementations of PlanManager. $C1$ guaranteed the satisfaction of its post-condition. $C2$ violated the subformula of the post-condition aimed to ensure the satisfaction of $P5$. To ensure the satisfaction of the post-condition we changed the sub-controller to obtain the lock over *condList* before accessing the variables of the *execCondChecker*.

Integration. We integrated each sub-controller into the original model of the *Executive* component and used the *model checker* to verify the resulting controller w.r.t. the properties $P1$-$P5$. As expected (and formally guaranteed by our framework), the resulting controller satisfies them. This confirms that FIDDLE supports verification-driven iterative and distributed development of controllers.

Threats to validity. The first threat to validity concerns the simulation of a "real" design process via reverse engineering of an existing controller model. The use of the original developer comments worked as a mitigation for this since some of the decisions we based on them and hence on the original design process. The second threat is that we used a single example to prove the effectiveness. However, the considered example is a medium size complex real case study [23], [24], [25], able to provide good feedback on FIDDLE. Finally, we analyze the considered example with a single box. Having multiple boxes would not affect the design process since boxes are distributed and analyzed in parallel.

B. Assessing scalability

To answer RQ.2, we set up two experiments in which we compared the performance of the *well-formedness* and the *substitutability checker* w.r.t. classical model checking as the size of the controller and the environment grew[2]. Our experiments are based on a set of *randomly-generated* models as is commonly done in software engineering [26], [27], [28], [29], [30]. In the first experiment, we generated an LTS model of the environment and its (complete) controller. We checked the system w.r.t. to a property of interest using a standard model checking procedure. Then, we marked one of the states of the controller as a black box, defined pre- and post- conditions for it and ran the well-formedness checker, comparing performance of the two. As our goal was to measure scalability as the number of states of the controller increases, we limited our analysis to the case of a single box. Indeed, the well-formedness checker works on the pre-condition of a black box at the time, after transforming all the post conditions of the black boxes in LTSs and plugging them instead of the corresponding box. Of course the number of boxes increases the number of states on which the well-formedness algorithm works, but 1) in practice the size of the post-condition, which determined the size of the corresponding LTL, is usually much smaller than size of the overall controller, and 2) also the size of the final controller increases with the number of boxes, since each box is substituted with a sub-controller (and, hence, also the model checker works on a much bigger model).

In the second experiment, we created a sub-controller from the complete controller which contains half of its states and

[2]The realizability checker and the model checker are not considered in the scalability evaluation since they reduce to standard model checking.

	#ContStates					
#EnvStates	100	120	140	160	180	200
10 000	2.46	2.53	2.26	2.28	2.28	2.26
12 000	2.69	2.55	2.10	1.93	1.47	1.79
14 000	2,75	2,80	2.86	2.67	2.01	2.54
16 000	3,51	3,44	3.29	2.54	2.94	2.37
18 000	3,46	3,28	3.04	3.07	3.02	2.80
20 000	3.35	3.68	3.52	3.41	3.40	3.30

TABLE I: Well-formedness vs. model checker $(T_w)/(T_m)$.

	#ContStates					
#EnvStates	100	120	140	160	180	200
10 000	0.30	0.22	*NS*	*NS*	*NS*	*NS*
12 000	0.31	0.25	0.21	*NS*	*NS*	*NS*
14 000	0.28	0.23	0.19	0.17	0.09	*NS*
16 000	0.30	0.26	0.19	0.16	0.13	0.08
18 000	0.34	0.24	0.19	0.16	0.15	0.11
20 000	0.33	0.23	0.20	0.14	0.13	0.12

TABLE II: Substitutability vs. model checker $(T_s)/(T_m)$.

the transitions between these states. We defined pre- and post-conditions for the sub-controller and ran the substitutability checker, comparing its performance with model-checking.

Experimental setup: We implemented a *random model generator* to create LTSs with a specified number of states, transition density (transitions per state) and number of events.

Environment. We generated environments of different sizes. We considered 10000 states (#EnvStates) and performed increases of 20% until 20000 states. We have chosen a transition density of 15, which corresponds to the one of the Mars Rover example. To simulate increments in the number of events, we set them to $\lfloor 110 \times \text{\#EnvStates}/10^4 \rfloor$.

Controller. We started with controllers with 100 states (#ContStates) and performed increases of 20% until 200 states. The transition density is 22 as in the *Executive* component of the Mars Rover example. The number of events is $\lfloor 0.82 \times \text{\#EnvStates} \rfloor$, where 0.82 is the ratio between the number of controller and environment events of the Mars Rover. One of the states of the LTS is marked as a black box, with 25% of the events of the controller in its interface.

To produce the sub-controller for the second experiment, we extracted it from the generated controller by taking half of the controller states and the transitions between them. We also added a new state s_f as a final state of the sub-controller and connected to it all the states of the sub-controller with a successor not included in the sub-controller.

Properties of interest, pre- and post-conditions. We present our scalability analysis considering $\Box(Q \rightarrow P)$, where Q and P are appropriate fluents, as property to be verified by the model checker and as pre-condition of the box in the controller. The post-condition is set as $\Box(\neg Q)$.

Methodology. We ran each experiment 5 times in an DS2_V2 Azure Machine, with 7GB of RAM, 2 CPU cores, 14GB Local SSD disk. For the first experiment, we computed the average time required by the well-formedness checker (T_w) and by the model checker (T_m) for each combination of the values of the #EnvStates and #ContStates, and the ratio between them. For the second, we measured the time required by the substitutability checker (T_s) and by the model checker (T_m).

Results for Experiment 1. Table I presents the results of our evaluation. It shows that, given an environment, well-formedness checker scales as well as a classical model checking as the number of controller states grows. For example, given an environment with 10 000 states, well-formedness checking was 3 times slower than classical model checking. However, this ratio did not change as the number of the states of the controller increased.

Results for Experiment 2. Table II shows that, given an environment, substitutability checking scales as classical model checking as the size of the controller increases. For example, given an environment with 10 000 states, substitutability checking was 3 time faster than model checking. Indeed, as the size of the controller increases, the model checker analyzes the final (complete) model of the controller, while the substitutability checker only consider the portion encapsulated into the sub-controller. The cases marked in Table II with *NS* represent those in which the substitutability checker returns a counterexample and hence are not significant for this analysis.

Threats to validity. Our scalability analysis is performed only on a single property. In order to mitigate this threat, we analyzed the scalability using another property, $\Diamond(P \rightarrow \Box Q)$, with post-condition $\Box(P \wedge Q)$. The obtained results were in line with the ones presented above. A more general analysis would study scalability on multiple properties obtained from a repository such as [31].

VIII. Related Work

The work closest to us is Sketch [5], which supports programmers in describing an initial structure of the program that can be completed using synthesis techniques, but, to the best of our knowledge, the same idea has not been applied for supporting the development of controllers. In the following, we survey approaches for producing correct controllers in the presence of partial information and incremental development.

1. Modeling partiality. Different formalisms, such as Modal Transition Systems (MTSs) [32], Partial Kripke Structures (PKSs) [8], $\mathcal{X}$KSs [9] and LTS$^{\uparrow}$ [21], support the specification of incomplete concurrent systems. Some of them could be used in an iterative development process although they have not been designed with this goal in mind and hence their application might not be straightforward. Other formalisms, such as Hierarchical State Machines (HSMs) [33], are used to model sequential processes via a top-down development process. However, HSMs can only be analyzed when a fully specified model is available.

2. Checking partial models. Many approaches to analyze partial models have been proposed, e.g., [34], [9], [8], and [33]. None of these technique are applicable in our approach where the missing sub-controllers are specified through pre- and post- conditions. [21] analyzes the assumption generation problem for Labeled Transition System with an additional interface operator, which describes how the system model interacts with the environment. This work is complementary

to ours and can be integrated to automatically generate post-conditions for the still undefined sub-controllers.

3. Substitutability checking. The goal of substitutability checking is to verify whether a possibly partial sub-controller can be plugged into a higher level structure without affecting its correctness. Problems such as "compositional reasoning" [35], [36], [37], [37], "component substitutability" [38], and "hierarchical model checking" [33], [39] are related to this part of our work. Our work differs because we first guarantee that the properties of interest are satisfied in the initially defined partial controller and then we check that provided sub-controllers are suitable to be plugged in the initial controller.

4. Synthesis. Program synthesis [40], [11] aims at computing a model of the system that satisfies the properties of interest. Many techniques for synthesizing controllers have been proposed (e.g., [3], [11], [41], [42], [43], [44]) and fully automated synthesis of highly non-trivial controllers of sizes over 2000 states is becoming possible [45] in special cases (e.g., limiting the types of synthesizable goals and using heuristics). However, such cases might not be applicable in general. We do not consider our approach to be alternative to synthesis, but instead integrate synthesis techniques to generate parts of the design, if applicable.

IX. Conclusion

We presented a verification-driven methodology, called FIDDLE, to support iterative distributed development of controllers. Although controller synthesis is making a lot of progress and can provide a viable solution in specific cases, we explored a complementary approach that tries to empower engineers with a way to design complex controllers that enables to recursively decompose a controller into a set of sub-controllers that can be specified in a way that correctness of the overall controller is ensured. Development of sub-controllers that satisfy their specifications can then be done independently, through distributed development. Alternatively, it may be done by reusing existing sub-controllers or through use of existing third-party services. It may also be done by synthesizing suitable sub-controllers.

Our preliminary validation of FIDDLE is promising. The approach would naturally fit into modern iterative, agile developments, further enriching them with the benefit of formal verification. In addition, while our current implementation is based on an explicit-state model-checker, we have been able to show that our incremental controller design and verification approach scales to realistic, large systems. Embedding the approach into efficient symbolic model checking would be the goal of future work, together with a more careful analysis of the practical benefits of the approach by applying it to developing real controllers.We also plan to extend the work to deal with both controllable and non-controllable environment events and to combine it with our previous work [46] to use verification to automatically generate post-conditions.

Acknowledgements

We thank Dimitra Giannakopoulou for her help with the Mars Rover case study.

References

[1] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software Engineering for Automotive Systems: A Roadmap," in *Proc. of FOSE'07*. IEEE Computer Society, 2007, pp. 55–71.

[2] J.-B. Nivoit, "Issues in Strategic Management of Large-Scale Software Product Line Development," Master's thesis, MIT, USA, 2013.

[3] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of Partial Behavior Models from Properties and Scenarios," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 384–406, 2009.

[4] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer, "Component Verification with Automatically Generated Assumptions," *J. Automated Software Engineering*, vol. 12, no. 3, pp. 297–320, 2005.

[5] A. Solar-Lezama, "Program Synthesis by Sketching," Ph.D. dissertation, University of California, Berkeley, 2008.

[6] ——, "Program sketching," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 475–495, 2013.

[7] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Controllability in Partial and Uncertain Environments," in *Proc. of ACSD'14*. IEEE, 2014, pp. 52–61.

[8] G. Bruns and P. Godefroid, "Model Checking Partial State Spaces with 3-Valued Temporal Logics," in *Proc. of CAV'99*, ser. LNCS, vol. 1633, 1999, pp. 274–287.

[9] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel, "Multi-Valued Symbolic Model-Checking," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 4, pp. 371–408, 2003.

[10] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso, "Planning and Monitoring Web Service Composition," in *Proc. of AIMSA'04*. Springer, 2004, pp. 106–115.

[11] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesising Non-anomalous Event-Based Controllers for Liveness Goals," *ACM Tran. Softw. Eng. Methodol*, vol. 22, 2013.

[12] E. Sandewall, *Features and Fluents (Vol. 1): The Representation of Knowledge about Dynamical Systems*. Oxford University Press, Inc., 1995.

[13] D. Calvanese, G. De Giacomo, and M. Y. Vardi, "Reasoning about Actions and Planning in LTL Action Theories," in *Proc. of KR'02*, vol. 2, 2002, pp. 593–602.

[14] D. Giannakopoulou and J. Magee, "Fluent Model Checking for Event-Based Systems," in *Proc. of SIGSOFT FSE'03*. ACM, 2003, pp. 257–266.

[15] W. Heaven, D. Sykes, J. Magee, and J. Kramer, "A Case Study in Goal-Driven Architectural Adaptation," in *Proc. of SEAMS'09*, ser. LNCS, vol. 5525, 2009, pp. 109–127.

[16] R. M. Keller, "Formal Verification of Parallel Programs," *Commun. ACM*, vol. 19, no. 7, pp. 371–384, 1976.

[17] R. Miller and M. Shanahan, "The Event Calculus in Classical Logic – Alternative Axiomatizations," *Elect. Trans. on AI*, vol. 4, 1999.

[18] G. De Giacomo and M. Y. Vardi, "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces," in *Proc. of IJCAI'13*. ACM, 2013, pp. 854–860.

[19] G. De Giacomo, R. De Masellis, and M. Montali, "Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness," in *Proc. of AAAI'14*, 2014, pp. 1027–1033.

[20] J. Magee and J. Kramer, *State Models and Java Programs*. Wiley, 1999.

[21] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer, "Assumption Generation for Software Component Verification," in *Proc. of ASE'02*. IEEE, 2002, pp. 3–12.

[22] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, "Learning Assumptions for Compositional Verification," in *Proc. of TACAS'03*. Springer, 2003, pp. 331–346.

[23] L. Levy, *Taming the Tiger: Software Engineering and Software Economics*, ser. Springer Books on Professional Computing Series. Springer-Verlag, 1987.

[24] S. M. S. Ltd, ""small project", "medium-size project", and "large project": What do these terms mean?" http://www.totalmetrics.com/function-points-downloads/Function-Point-Scale-Project-Size.pdf, 2004.

[25] S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis, "Testing conformance of real-time applications by automatic generation of observer," in *Proc. of the 4th Workshop on Runtime Verification (RV 2004)*. Electronic Notes in Theoretical Computer Science, 1998, pp. 23–43.
[26] D. Tabakov and M. Y. Vardi, "Experimental Evaluation of Classical Automata Constructions," in *Proc. of LPAR'05*. Springer, 2005, pp. 396–411.
[27] M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin, "Antichains: A New Algorithm for Checking Universality of Finite Automata," in *CAV'06*. Springer, 2006, pp. 17–30.
[28] D. Tabakov and M. Y. Vardi, "Model Checking Buchi Specifications," in *Proc. of LATA'07*, 2007, pp. 565–576.
[29] P. Saadatpanah, M. Famelis, J. Gorzny, N. Robinson, M. Chechik, and R. Salay, "Comparing the Effectiveness of Reasoning Formalisms for Partial Models," in *Proc. of MoDeVVA'12*. ACM, 2012, pp. 41–46.
[30] M. Famelis, R. Salay, and M. Chechik, "Partial Models: Towards Modeling and Reasoning with Uncertainty," in *Proc. of ICSE'12*. IEEE, 2012, pp. 573–583.
[31] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property Specification Patterns for Finite-State Verification," in *Proc. of FMSP'98*. ACM, 1998, pp. 7–15.
[32] K. G. Larsen and B. Thomsen, "A Modal Process Logic," in *Proc. of LICS'88*. IEEE, 1988, pp. 203–210.
[33] R. Alur and M. Yannakakis, "Model Checking of Hierarchical State Machines," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 23, no. 3, pp. 273–303, May 2001.
[34] M. Huth, "Model Checking Modal Transition Systems Using Kripke Structures," in *Proc. of VMCAI'02*, ser. LNCS, vol. 2294, 2002, pp. 302–316.
[35] C. B. Jones, "Tentative Steps Toward a Development Method for Interfering Programs," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 5, no. 4, pp. 596–619, 1983.
[36] R. Alur and T. Henzinger, "Reactive Modules," *Formal Methods in Software Design*, vol. 15, no. 1, pp. 7–48, 1999.
[37] A. Pnueli, "In Transition from Global to Modular Temporal Reasoning about Programs," in *Logics and Models of Concurrent Systems*, K. R. Apt, Ed. Springer-Verlag New York, Inc., 1985, pp. 123–144.
[38] S. Chaki, E. M. Clarke, N. Sharygina, and N. Sinha, "Verification of Evolving Software via Component Substitutability Analysis," *Formal Methods in Software Design*, vol. 32, no. 3, pp. 235–266, 2008.
[39] R. Alur and M. Yannakakis, "Model Checking of Hierarchical State Machines," *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 6, pp. 175–188, 1998.
[40] A. Pnueli and R. Rosner, "On the Synthesis of a Reactive Module," in *Proc. of POPL'89*. ACM, 1989, pp. 179–190.
[41] A. Cimatti and S. Tonetta, "Contracts-Refinement Proof System for Component-Based Embedded Systems," *Science of Computer Programming*, vol. 97, pp. 333–348, 2015.
[42] O. Kupferman and M. Vardi, "Synthesis with Incomplete Information," in *Advances in Temporal Logic*. Springer, 2000, pp. 109–127.
[43] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, "Weak, Strong and Strong Cycling Planning via Symbolic Model Checking," IRST, Tech. Rep., 2015.
[44] K. Akesson, M. Fabian, H. Flordal, and R. Malik, "Supremica – An Integrated Environment for Verification, Synthesis and Simulation of Discrete Event Systems," in *Int. Workshop on Discrete Event Systems*. IEEE, 2006, pp. 384–385.
[45] D. Ciolek, V. A. Braberman, N. D'Ippolito, and S. Uchitel, "Technical Report: Directed Controller Synthesis of Discrete Event Systems," *CoRR*, vol. abs/1605.09772, 2016.
[46] C. Menghi, P. Spoletini, and C. Ghezzi, "Dealing with incompleteness in automata-based model checking," to appear in FM 2016: Formal Methods, 2016.