https://doi.org/10.1007/s00165-019-00484-1
The Author(s) © 2019
Formal Aspects of Computing

Formal Aspects of Computing

A verification-driven framework for iterative design of controllers

Claudio Menghi[1], Paola Spoletini[2], Marsha Chechik[3], and Carlo Ghezzi[4]

[1]University of Luxembourg, Luxembourg, Luxembourg
[2]Kennesaw State University, Marietta, USA
[3]University of Toronto, Toronto, Canada
[4]Politecnico di Milano, Milan, Italy

Abstract. Controllers often are large and complex reactive software systems and thus they typically cannot be developed as monolithic products. Instead, they are usually comprised of multiple components that interact to provide the desired functionality. Components themselves can be complex and in turn be decomposed into multiple sub-components. Designing such systems is complicated and must follow systematic approaches, based on recursive decomposition strategies that yield a modular structure. This paper proposes FIDDle–a comprehensive verification-driven framework which provides support for designers during development. FIDDle supports hierarchical decomposition of components into sub-components through formal specification in terms of pre- and post-conditions as well as independent development, reuse and verification of sub-components. The framework allows the development of an initial, partially specified design of the controller, in which certain components, yet to be defined, are precisely identified. These components can be associated with pre- and post-conditions, i.e., a contract, that can be distributed to third-party developers. The framework ensures that if the components are compliant with their contracts, they can be safely integrated into the initial partial design without additional rework. As a result, FIDDle supports an iterative design process and guarantees correctness of the system at any step of development. We evaluated the effectiveness of FIDDle in supporting an iterative and incremental development of components using the K9 Mars Rover example developed at NASA Ames. This can be considered as an initial, yet substantive, validation of the approach in a realistic setting. We also assessed the scalability of FIDDle by comparing its efficiency with the classical model checkers implemented within the LTSA toolset. Results show that FIDDle scales as well as classical model checking as the number of the states of the components under development and their environments grow.

Keywords: Distributed development; Controller design; Verification-driven development.

Correspondence and offprint requests to: C. Menghi, E-mail: claudio.menghi@uni.lu

Published online: 05 June 2019

1. Introduction

Software systems are usually comprised of multiple components—portions of the system that provide a desired functionality. In large and complex systems, components themselves can be complex, and decomposed into multiple sub-components. Hence, especially when large systems are considered, system design must follow systematic approaches, based on recursive decomposition strategies that support the development of modular structures. A good decomposition and a careful specification should allow sub-components to be developed in isolation by different members of the development team or be delegated to third parties [Par72a, Par72b]. It should also be possible to reuse off-the-shelf components or delegate development of parts of the system to external service providers [PBKS07, PBvDL05, ABKS16, CH01]. In essence, software development can often be viewed as a distributed endeavor, where different decentralized developers (internal engineers, subcontractors, component and service providers) are coordinated by the organization responsible for the entire system. The main problems of distributed development are in the integration phase, when separately validated components are composed with the other parts of the system and checked for overall correctness. In practice, software failures in the integration phase may lead to expensive and painful changes that may affect the components, the rest of the system, and even lead to changes in the modular structure.

In this article, we focus on the iterative and incremental design of components that abstractly behave as controllers. This term is commonly used to denote components that interact with a complex environment—which may include both physical devices and humans—through events. The controller processes the events generated by the environment and generates events to control it. In particular, we focus on the *provably correct iterative and incremental development of complex, modular controllers*. By this we mean that that component integration is safe and does not require any rework. It guarantees overall correctness by construction.

Recently, several researchers [vBFH+14, tBRdV16] focused on synthesis as a way to obtain correct-by-construction components. These techniques are tremendously useful for in-house development of small components or development of individual components. Yet, they are not appropriate for many real-world scenarios, due to their inability to support *iterative and incremental* development which becomes necessary when the problem is too large to be handled in a single step. In addition, they do not support component reuse. For these reasons, although significant advances have been made in the direction of synthesis, the fully automated approach based on synthesis is not yet (and perhaps will never be) able to produce complex components. A more viable solution is to use synthesis activities in the design process as a *support to human effort* [SL08, SL13]. On the other side of the spectrum, approaches presented in the literature [BG99, CDEG03, MSG16, MSG17, BMS+17] provide support for model-checking and top-down refinement of partial models with the goal of preserving correctness. However, while these techniques guarantee correctness at each development stage, they do not explicitly address the problem of decentralized development and bottom-up integration, which are needed to support iterative and incremental development of real-world components. This is the problem we tackle in this work in a detailed and actionable manner.

We believe that the complete automation of complex components development is both impossible and undesired. Rather, we envision a systematic and formally verified design process where the construction of the overall structure of a complex controller and the various development steps require insight and experience which are human, not machine characteristics [SL08]. 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 provide several types of support:

S1. *Modeling support for design activities performed by humans.* Novel tools for software development should help humans with modeling formalisms that effectively support components design. This includes support in the contexts in which software components are produced by first creating an initial partial high-level model of a component, where portions of the system that should be later defined are clearly identified, and then developing a detailed model of the behavior of the system for the partially specified portions. The tools should support (i) the description of the environment in which the component will be deployed, (ii) the specification of its required properties, (iii) the creation of the initial (partial) structure of the component, and (iv) the design of the component behavior in the portion of the component that are left unspecified.

S2. *Model analysis support.* As developers produce models of their components, they require automatic support to check their designs. Such support should (j) check whether the properties of interest can be satisfied by refining unspecified parts; (jj) check whether the initial partial design satisfies a set of properties of interest. The first check allows developers to verify whether it is possible to refine unspecified parts in a way that guarantees the satisfaction of the property of interest. This is a necessary requirement for enabling distributed development. The second check is performed after the contracts of the unspecified parts are defined. It allows the verification of whether respecting the given contracts guarantees satisfaction of the properties of interest. If the answer to the latter question is negative, further information that may help the user in fixing the errors, e.g., some form of counterexamples, should be provided.

S3. *Support for iterative and incremental development of unspecified parts.* When complex components are considered, it is desirable to first define a high level behavior of the component, leaving some parts temporarily unspecified, and then develop the unspecified parts or reuse existing off-the-shelf sub-components, or even delegate provision of the functionality to an external service provider. To enable this type of development, there is a need for developers to precisely describe what the unspecified parts should do.

S4. *Support for integrating (sub-)components.* When (sub-)components associated with the unspecified parts are created and delivered, they must be integrated in the original partially specified design. *Integration* is a process that replaces the unspecified parts with their actual design. A desired property of the framework we aim to develop is that *no work (manual or automatic) should be performed at the integration level*.

In this paper, we propose a unified framework called FIDDle (a Framework for Iterative and Distributed Design of components). FIDDle provides a set of tools and techniques for iterative and incremental verification-driven component development. The framework is not based on any specific component models, like the ones commonly adopted by industry (such as Java Beans, Microsoft's COM family, or others). It is positioned at a higher and more abstract level, and assumes components to be fragments of functionality modeled as state machines. The framework supports a formal specification of global properties, a decomposition process and a specification of component interfaces by providing a set of tools to guarantee correctness of the different artifacts produced during the process. The capability of FIDDle to approximate complex designs by contracts and to replace a prototypical implementation with a more sophisticated one at a later stage fosters decomposition of complex designs and component reuse. Specific novel contributions of FIDDle are:

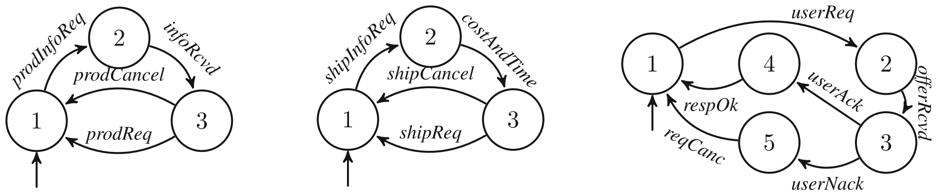
1. a new formalism, called *Interface Partial Labeled Transition System (IPLTS)*, for specifying components through a decomposition that encapsulates sub-components into unspecified black-box states;
2. an approach to specify *the expected behavior of black-box states* via pre- and post-conditions expressed in Fluent Linear Time Temporal Logic (FLTL); and
3. a notion of *component correctness* and a *local verification procedure* that *guarantees preservation of global properties* once the components are composed.

We also report on the evaluation of FIDDle on a realistic case study obtained by reverse-engineering the executive module of the Mars Rover developed at NASA [GPB02, CGP03, GPB05]. Scalability is evaluated by considering randomly-generated examples.

FIDDle was first presented in [MSCG18]. This paper extends the work in [MSCG18] in several directions:

- it presents a complete formal treatment of the work, including proofs of all lemmas and theorems;
- it provides a detailed description of the algorithms;
- it shows how the presented approach can be applied recursively;
- it provides a thorough comparison with related work;
- it describes the FIDDle tool support;
- it provides additional detail of our evaluation.

The rest of the paper is organized as follows. Section 2 describes our running example. Section 3 provides an overview of FIDDle. Section 4 gives the necessary background. Section 5 describes the semantics for FLTL on finite traces. Section 6 presents Interface Partial Labeled Transition Systems. Section 7 defines a set of algorithms for reasoning on partial components and describes their implementation. Section 8 reports on an evaluation of the effectiveness and scalability of the proposed approach. Section 9 compares FIDDle with related approaches, and Sect. 10 concludes the paper.



(a) Furniture-sale. **(b)** Shipping. **(c)** User.

P1: ship and product info are provided only if a request has been received.
P2: when user requests are processed, offers are considered only after users received information about the desired product.
P3: the furniture service is activated only if the user has decided to purchase.
P4: when a user request is canceled by the p&d system, no user ack precedes the cancellation.

(d) Properties of the p&d system.

Fig. 1. The p&d environment (Figs. 1a, 1b, 1c) and the properties the p&d system must ensure (Fig. 1d)

2. Running example

We illustrate FIDDle using a simple example of *purchase&delivery* (p&d) [PBB+04, DBPU13], shown in Fig. 1. The p&d system supports furniture purchase and delivery. The system allows users to check whether certain items are present, and to order the desired product or cancel the order.

To provide this functionality, the p&d system uses two existing web services, which implement the furniture-sale and the shipping. The behavior of the furniture-sale service is described in Fig. 1a through a simple state machine. The service is initially in the state 1. The transition labeled with the action *prodInfoReq* allows the furniture-sale component to be queried to check the presence of some furniture. The transition labeled with the action *infoRcvd* indicates that the information regarding the furniture is provided. Finally, transitions labeled with *prodReq* and *prodCancel* indicate that the furniture is requested or the order is canceled.

The behavior of the shipping service is described in Fig. 1b. The transition labeled with *shipInfoReq* allows the shipping component to be queried to check the presence of some furniture. The transition labeled with *costAndTime* indicates that the information regarding the cost and time of the delivery is provided. Transitions labeled with *shipReq* and *shipCancel* indicate that the shipping is being requested and that the order is canceled, respectively.

The desired interactions of the users with the yet-to-be-defined component are described in Fig. 1c. The transition labeled with *userReq* indicates that the user performs a request to the system. The transition labeled with *offerRcvd* indicates that the user receives the desired information (furniture presence and shipping information). Based on this information, the user can choose whether to accept or refuse the offer. This is represented through the transitions labeled with the actions *userAck* and *userNack*. The system then has to confirm to the the user that her choice has been correctly processed via transitions labeled with the actions *respOk* and *reqCanc*.

The goal of the development team is to design a component, referred in the following as *the p&d component*, which acts as a controller interacting with the user and the two services in order to satisfy the user requests. The furniture-sale, the shipping service and the user represent the environment within which the component under development should operate. The final design of the p&d component must ensure that the controller, when plugged into its environment, will satisfy a set of properties of interest, reported in Fig. 1d. For example, in the final system, ship and product info should not be provided to the user if a request has not been issued. The development team decides to adopt a top-down development approach, which first creates an initial, partial and preliminary high-level description of the system where the still-to-be-refined parts of the system are explicitly indicated. The unspecified parts of the component are refined into sub-components, whose design and implementation are delegated to third-party developers, who receive a contract that should be satisfied by the sub-component. It is strongly desired that *no rework* be performed during the integration phase.

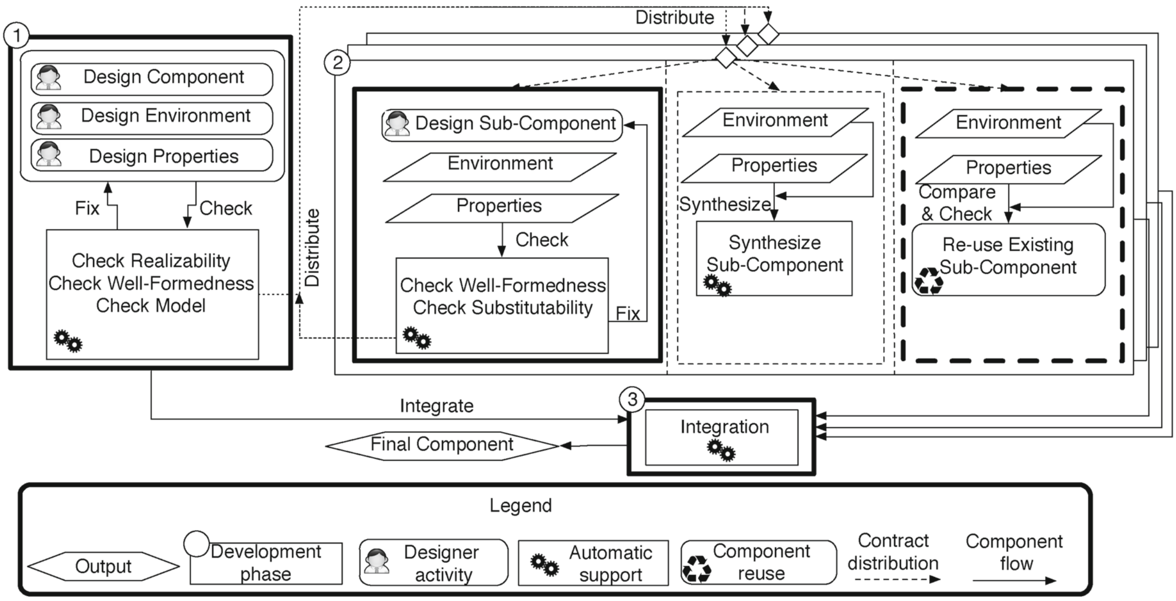


Fig. 2. Overview of the application of FIDDle for developing a component. Thick-bordered components are implemented in FIDDle. Thick-dashed bordered components are currently supported by the theory presented in this paper, but they are still not fully implemented. Thin-dashed bordered components are not discussed in this work

3. Overview

FIDDle is a verification-driven environment supporting iterative and incremental controller developments, as described in Sect. 1. A high-level view of FIDDle is shown in Fig. 2. FIDDle allows controllers to be modularly and incrementally developed in a distributed manner, through a set of development phases in which the human insight and experience are exploited to achieve a verified modular structure (rounded boxes labeled with a designer icon and a recycle symbol indicate design or reuse, respectively) and phases in which automated support is provided (squared boxes labeled with a pair of gearwheels). Automatic support is provided to verify the current state of the design, integrate synthesized or off-the-shelf components, decentralize component development to third parties, and check whether integrated components correctly fit into the overall design. FIDDle structures controller developments according to a set of phases described in the following.

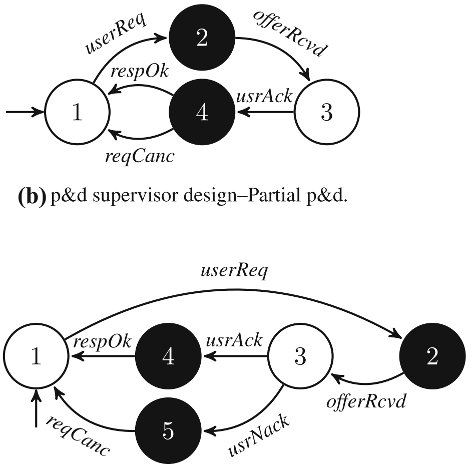
3.1. Creating an initial component design

This phase is identified in Fig. 2 with the symbol ①. FIDDle provides modeling support for design activities performed by humans and analysis support for design validation.

FIDDle provides suitable formalisms to specify requirements and to model behavior of the controller and of its (sub-)components (step `S1` in Sect. 1). The development team formalizes the component's desired properties through the requirements specification language and designs an initial, high-level structure of the component. Components are modeled using a state-based formalism that can clearly identify sub-components, represented as *black-box* states, whose internal design is delayed to a later stage or split apart for distributed development by other parties. In the following, we refer to other (non black-box) states as "regular". Black-box states are enriched with an *interface* that provides information on the universe of events relevant to the black box. They are also decorated with pre- and post-conditions that allow distributed teams to develop sub-components without the need to know about the rest of the system. The *contract* of a black-box state consists of its interface and the pre- and post-conditions.

$$\boxed{\begin{array}{l}
P1 = (\neg((\neg \mathit{F\_UserReq})\,\mathcal{U}\,(\mathit{F\_ShipInfoReq} \vee \mathit{F\_ProdInfoReq})))\\
P2 = \Box(\mathit{F\_UserReq} \rightarrow (\neg((\neg \mathit{F\_InfoRcvd})\,\mathcal{U}\,\mathit{F\_OfferRcvd})))\\
P3 = \Box(\mathit{F\_UsrReq} \rightarrow ((\neg((\neg \mathit{F\_UserAck})\,\mathcal{W}\,\mathit{F\_ShipReq}))\\
P4 = \Box((\mathit{F\_UsrReq} \wedge ((\neg \mathit{F\_UsrReq})\,\mathcal{U}\,\mathit{F\_ReqCanc})) \rightarrow ((\neg \mathit{F\_UserAck})\,\mathcal{U}\,\mathit{F\_ReqCanc}))
\end{array}}$$

(a) FLTL formulation of the p&d properties.

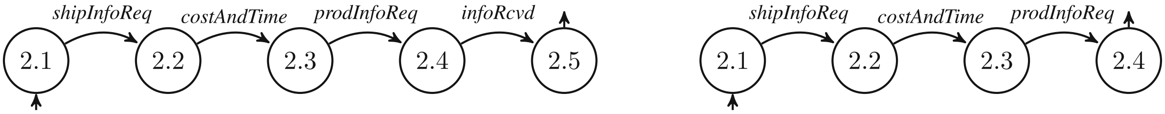


(b) p&d supervisor design–Partial p&d.

(d) Another partial p&d component.

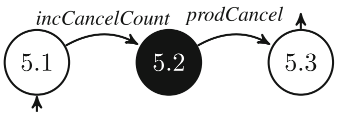
	State 2
interface	{ *prodInfoReq*, *infoRcvd*, *shipInfoReq*, *costAndTime* }
pre	$\Diamond(\mathit{F\_UserReq} \wedge \neg \Diamond(\mathit{F\_RespOk} \vee \mathit{F\_ReqCanc}))$
post	$(\Diamond \mathit{F\_InfoRcvd}) \wedge (\Diamond \mathit{F\_CostAndTime})$
	State 4
interface	{ *prodReq*, *shipReq* }
pre	$\Box(\mathit{F\_UserReq} \rightarrow \Diamond \mathit{F\_InfoRcvd})$
post	$\Diamond(\mathit{F\_ProdReq}) \wedge \Diamond(\mathit{F\_ShipReq})$
	State 5
interface	{ *prodCancel*, *shipCancel* }
pre	$\Box(\mathit{F\_UserReq} \rightarrow \Diamond \mathit{F\_InfoRcvd})$
post	$\Diamond(\mathit{F\_ProdCancel}) \wedge \Diamond(\mathit{F\_ShipCancel})$

(c) Contracts for black-box states of Figs. 3b, 3d and 3j.



(e) A sub-component for black-box state 2.

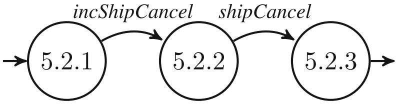
(f) Another sub-component for black-box state 2.



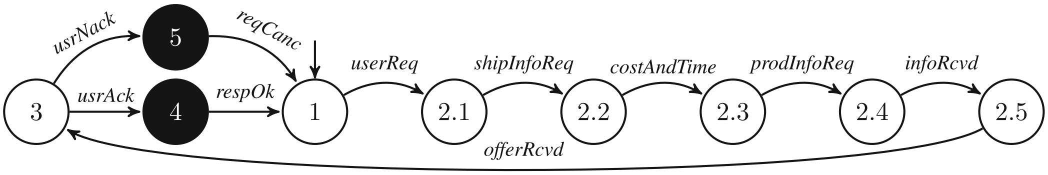
(g) A sub-component for black-box state 5.

	State 5.2
interface	{ *shipCancel* }
pre	$\Box(\mathit{F\_UserReq} \rightarrow \Diamond \mathit{F\_InfoRcvd})$
post	$\Diamond(\mathit{F\_ShipCancel})$

(h) Contract for black-box state 5.2.



(i) A sub-component for black-box state 5.2.



(j) Integration of the sub-component of Fig. 3e and the component of Fig. 3d.

Fig. 3. The p&d running example: artifacts produced by FIDDle

In the p&d example, the environment (assumed as given) in which the p&d component will be deployed is composed by the furniture-sale component (Fig. 1a), the shipping component (Fig. 1b) and the user (Fig. 1c). A possible initial design for the p&d component is shown in Fig. 3b. It contains the regular states 1 and 3 and black-box states 2 and 4. State 1 is the initial state. Whenever a *userReq* event is detected, the component moves from the initial state 1 into the black-box state 2, which represents a sub-component in charge of managing the user request. An event *offerRcvd* which indicates that an offer is provided to the user labels the transition to state 3. The interface of the black-box states 2 and 4 and their pre- and post-conditions are shown in Fig. 3c. The interface indicates that events *prodInfoReq*, *infoRcvd*, *shipInfoReq* and *costAndTime* can occur while the component is in the black-box state 2. Pre- and post-conditions need to be provided by the developers. Pre-conditions specify properties that hold up to entering the corresponding black-box state and post-conditions specify properties that should be ensured by black-box states. Both of them can be used in the analysis: whether the finite behaviors reaching the black-box state guarantee satisfaction of the pre-conditions and whether the black-box state can be replaced by their post-conditions to verify the validity of properties of interest. The pre- and post-conditions are discussed in detail in this section.

FIDDle supports the designer by checking properties of the design at different stages of development (step `S2` in Sect. 1). The *realizability checker* confirms the existence of a realizable component that can be integrated to complete a partially specified component and ensures satisfaction of the properties of interest. If such a component does not exist, the designer needs to redesign the partially-specified component. The *well-formedness checker* verifies that both the pre- and the post-conditions of black-box states are satisfiable. Finally, the *model checker* verifies whether the (partial) component (together with its contract) guarantees satisfaction of the properties of interest.

In the p&d example, the model checker identifies a problem with the partial solution sketched in Fig. 3b. No matter how the black-box state 2 is to be defined, the p&d component cannot satisfy property *P4* since every time *reqCanc* occurs it is preceded by *usrAck*. This suggests a re-design of the p&d component, which may lead to a new model, shown in Fig. 3d. This model includes two regular states: state 1, in which the component waits for a new user request, and state 3, in which the component 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-box states, to be refined later.

To support iterative and incremental development of unspecified parts, FIDDle allows designers to specify pre- and post-conditions for the black-box states (steps `S3` and `S4` in Sect. 1). In the p&d example, pre- and post-conditions of the black-box state 2 specify that there is a pending user request, and that cost, time and product information are collected. Pre- and post-conditions of the black-box state 4 specify that *infoRcvd* has occurred after the user request, and both a product and shipping requests are performed. Finally, pre- and post-conditions of the black-box state 5 specify that *infoRcvd* has occurred after the user request and before entering the state, and both the product and the shipping requests are canceled when leaving the state. This model is checked using the verification tools; since it passes all the checks, it can be used in the next phase of the development.

The design team may choose to refine the component or *distribute* the development of unspecified sub-components (represented by black box states) to other (internal or external) development teams. In both cases, the sub-component can be designed by only considering the contract of the corresponding black-box state. Each team can develop the assigned sub-component or reuse existing components.

3.2. Sub-component development

This phase is identified in Fig. 2 with the symbol ②. During sub-component development, FIDDle provides support to the modeling activities performed by humans (step `S1` in Sect. 1). In the p&d example, each team can design the assigned sub-component using any available technique, including manual design (left side), reusing of existing sub-components (right side) or synthesizing new ones from the provided specifications (center).

To support iterative and incremental development of unspecified parts and safe integration of the developed of sub-components, FIDDle requires the developed sub-components to satisfy some constraints, namely,

1. given the stated pre-condition, the sub-component has to satisfy its post-condition, and
2. the sub-component should operate in the same environment as the overall partially specified component.

Sub-component development can itself be an iterative process (see Sect. 3.4) but neither the model of the environment nor the overall properties of the system can be changed during this process. Otherwise, the resulting sub-component cannot be safely and automatically integrated into the overall system.

In the p&d example, development of the sub-component for the black-box state 2 is delegated to an external (third-party) contractor. Candidate sub-components are shown in Figs. 3e-3f. In the case of Fig 3e, the component requests shipping info details and waits until the shipping service provides the shipment cost and time. Then it queries the furniture-sale service to obtain the product info. In the case of Fig 3f, the shipping and the furniture services are queried, but the sub-component does not wait for an answer from the furniture-sale. Since these candidates are fully defined, the well-formedness check is not needed. Yet, the *substitutability checking* confirms that of these, only the sub-component in Fig. 3e satisfies the post-condition in Fig. 3c.

3.3. Integration of sub-components

This phase is identified in Fig. 2 with the symbol ③. FIDDle supports integration of the development of sub-components and guarantees that if each sub-component is developed correctly w.r.t. the contract of the corresponding black-box state, the component obtained by integrating the sub-components is also correct (step `S4` in Sect. 1). In the p&d example, the sub-component in Fig. 3e passes the substitutability check and can be a valid implementation of the black-box state 2 in Fig. 3d. Integration is shown in Fig. 3j.

3.4. Recursive application of FIDDle

FIDDle can be applied recursively, allowing to distribute development of portions of the sub-components (e.g., to third-party vendors). This is indicated in Fig. 2 with the symbol ②.

For example, Fig. 3g shows an initial partial design of the sub-component associated with state 5 of the p&d example. This partially-specified sub-component requires the system to increase the count of canceled orders, enters the black-box state 5.2, and finally cancels the order. FIDDle provides automated support for the sub-component design. The well-formedness confirms that the pre- and the post- conditions of black-box states are satisfiable. The substitutability checking notifies the designer that the sub-component in Fig. 2 does not ensure the satisfaction of its post-condition. Indeed, there is no guarantee that shipping of the product is canceled (i.e., the *ShipCancel* event occurs). The post-condition for the black-box state 5.2, shown in Fig. 3h, which ensures that the shipping is performed before the black-box state 5.2 is exited by the system, guarantees that the sub-component passes the substitutability checking. In turn, the black-box state 5.2 is decomposed into the sub-component represented in Fig. 3i. This component is completely specified. Upon its entry, the counter of the canceled shipping orders is incremented and then the shipping is canceled. When it is entered, the counter of the canceled shipping orders is incremented by one. The substitutability checking confirms that the sub-component is substitutable and can be integrated within the black-box state 5.2. This iterative process can be applied for any partially specified subcomponent.

To conclude, FIDDle allows the iterative and incremental distributed development of controllers and provides the support described in Sect. 1. FIDDle relies on an extension of Labeled Transition Systems (LTS) that allows the representation of incompleteness and offers the possibility of defining an initial modular incomplete structure. It also allows distributing the design of the missing components and offers automatic support for the verification of the different parts and for substitutability analysis of components within a partially defined structure.

4. Background

This section contains background knowledge and definitions used in the rest of the paper. Section 4.1 introduces LTS. Section 4.2 provides a high level description of Büchi automata (BA). Section 4.3 introduces Fluent Linear Time Temporal Logic (FLTL). Finally, Sect. 4.4 outlines a procedure for verifying the satisfaction of FLTL properties on LTS.

4.1. Labeled transition systems

Labeled Transition Systems are a specific type of state machines commonly used to model software components and their environments.

Definition 4.1 [Kel76] Let Act be the universal set of observable events and let τ be an unobservable local event. A *finite Labeled Transition System* (LTS$_f$) is a tuple $F = \langle Q, q_0, A, \Delta, Q_f \rangle$, where

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- $A \subseteq Act$ is a finite set of events,
- $\Delta \subseteq Q \times A \cup \{\tau\} \times Q$ is the transition relation, and
- $Q_f \in Q$ is the set of final states.

A *Labeled Transition System (LTS)* $L = \langle Q, q_0, A, \Delta \rangle$ is an LTS$_f$ where the set of final states is not defined.

For example, Fig. 3e contains an LTS$_f$with states 2.1, 2.2, 2.3, 2.4 and 2.5, where 2.1 and 2.5 are its initial and final states, respectively. Transitions specify how the LTS$_f$ evolves by firing transitions labeled with events *shipInfoReq*, *costAndTime*, *prodInfoReq* and *infoRcvd*.

Fig. 1a contains an LTS for modeling a furniture-sale component. The LTS is defined over the states 1, 2 and 3, where state 1 is initial. Transitions specify how the LTS evolves by firing transitions labeled with events *prodInfoReq*, *infoRcvd*, *prodCancel* and *prodReq*.

Given a state q (of an LTS or an LTS$_f$), let $\Delta(q)^-$ and $\Delta(q)^+$ denote its incoming and outgoing transitions, respectively, and let $\Delta(q)$ denote the union of the incoming and outgoing transitions of q. For example, in the LTS in Fig. 1a, $\Delta(1)^+ = \{(1, \mathit{prodInfoReq}, 2)\}$. Given a transition $\delta = (q_1, e, q_2)$, we use δ^- to indicate its source q_1, δ^+ to indicate its destination q_2, and δ_e for its label e. For example, for the transition $\delta = (2, \mathit{infoRcvd}, 3)$ of the LTS in Fig. 1a, $\delta^- = 2$, $\delta^+ = 3$ and $\delta_e = \mathit{infoRcvd}$.

Definition 4.2 Let $F = \langle Q, q_0, A, \Delta, Q_f \rangle$ be a LTS$_f$. A *finite* sequence $q_0, e_0, q_1, e_1, \ldots, q_n, e_n, q_{n+1}$, such that for every $0 \leq i \leq n$, $(q_i, e_i, q_{i+1}) \in \Delta$ and $q_{n+1} \in Q_f$, is a *finite* execution of F, and $\pi = e_0, e_1, \ldots, e_n$ is a *trace* of F. Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS. An *infinite* sequence $q_0, e_0, q_1, e_1, \ldots$, such that for every $i \geq 0$, $(q_i, e_i, q_{i+1}) \in \Delta$, is an *infinite* execution of L, and $\pi = e_0, e_1, \ldots$ is a *trace* of L.

Traces of an LTS$_f$ are finite and reach final states representing finite computations. For example, *shipInfoReq*, *costAndTime*, *prodInfoReq*, *infoRcvd* is the only finite trace of the LTS$_f$ in Fig. 3e. LTS traces are infinite, i.e., the system is not designed to stop. For example, *prodInfoReq*, *infoRcv*, *prodReq* repeated infinitely often is an infinite trace of the LTS in Fig. 1a.

Definition 4.3 Let $\pi = e_0, e_1, \ldots, e_n$ and $\pi' = e'_0, e'_1, \ldots, e'_n$ be two sequences of events. We use the notation $\pi; \pi'$ to indicate their concatenation $e_0, e_1, \ldots, e_n, e'_0, e'_1, \ldots, e'_n$.

In the rest of this paper, when not specified, we assume that LTS$_f$ and LTS are *minimized* with respect to bisimulation. Intuitively, given an LTS$_f$ (respectively, an LTS), the minimization procedure removes τ actions and generates a LTS$_f$ (respectively, an LTS) with the same behavior (for additional information about minimization, refer to [MK99]).

Parallel composition is a symmetric operator that takes two LTS$_f$ and computes a resulting one by synchronizing on shared events and interleaving the others.

Definition 4.4 Let $M = \langle Q_M, q_{0,M}, A_M, \Delta_M, Q_{f,M} \rangle$ and $N = \langle Q_N, q_{0,N}, A_N, \Delta_N, Q_{f,N} \rangle$ be LTS$_f$. *Parallel composition* of M and N (denoted by $M \parallel N$) is an LTS$_f$ $P = \langle Q_M \times Q_N, \langle q_M^0, q_N^0 \rangle, A_M \cup A_N, \Delta, Q_{f,M} \times Q_{f,N} \rangle$, where Δ is the smallest relation that satisfies the following rules:

$$(1)\quad \frac{(s, l, s') \in \Delta_M}{(\langle s, t \rangle, l, \langle s', t \rangle) \in \Delta}, \; l \in A_M \setminus A_N \text{ or } l = \tau$$

$$(2)\quad \frac{(t, l, t') \in \Delta_N}{(\langle s, t \rangle, l, \langle s, t' \rangle) \in \Delta}, \; l \in A_N \setminus A_M \text{ or } l = \tau;$$

$$(3)\quad \frac{(s, l, s') \in \Delta_M, (t, l, t') \in \Delta_N}{(\langle s, t \rangle, l, \langle s', t' \rangle) \in \Delta}, \; l \in A_N \cap A_M, l \neq \tau;$$

where $l \in A_M \cup A_N \cup \{\tau\}$, s and s' are states in Q_M, and t and t' are states of Q_N. Parallel composition of two LTS is defined identically, except that final states are excluded.

Rules (1) and (2) indicate that there is an interleaving on non-shared events and Rule (3) that there is a synchronization on shared ones.

Given a set of events H, the *hiding operator* turns all the transitions labeled with events in H into τ-labeled, i.e., unobservable events.

Definition 4.5 Let $F = \langle Q, q_0, A, \Delta, Q_f \rangle$ be an LTS$_f$ and $H \subseteq A$. $\backslash$ is a *hiding operator* such that $N = F \backslash H$ is an LTS$_f$ $N = \langle Q, q_0, A', \Delta', Q_f \rangle$, where the following conditions hold:

- $A' = A \setminus H$;
- $\Delta' = \{(q, e, q') \mid (q, e, q') \in \Delta \text{ and } e \in A \setminus H\} \cup \{(q, \tau, q') \mid (q, e, q') \in \Delta \text{ and } e \in H\}$.

For LTS, hiding is defined identically, except that final states are excluded.

For example, if the hiding operator is applied to the LTS in Fig. 1a w.r.t. the set {*prodInfoReq*}, the transition from state 1 to state 2 labeled with *prodInfoReq* is replaced by one labeled with τ.

4.2. Büchi automata

Büchi Automata are a reference formalism commonly used in the verification community.

Definition 4.6 [Büc90] Let AP be a finite set of atomic propositions. A *Büchi automaton* (BA) defined over AP is a tuple $B = \langle Q, q_0, A, \Delta, Q_a \rangle$ such that:

- Q is a finite set of states;
- q_0 is the initial state;
- $A \subseteq AP$ is a finite set of atomic propositions;
- $\Delta \subseteq Q \times A \times Q$ is the transition relation;
- $Q_a \subseteq Q$ is the set of the accepting states.

Given a BA $B = \langle Q, q_0, A, \Delta, Q_a \rangle$, $\pi = e_0, e_1, \ldots$ is an *infinite trace* of B if there exists an infinite sequence $q_0, e_0, q_1, e_1, \ldots$, where, for every $i \geq 0$, $(q_i, e_i, q_{i+1}) \in \Delta$ and there exists a state $q_i \in Q_a$ that appears in the sequence infinitely often.

It is possible to check (using a standard procedure described in [Büc90]) whether a given BA has no infinite trace. We indicate this check by a function CHECKEMPTINESS(B) which returns *true* if the automaton accepts no infinite traces and thus is empty, and *false* otherwise.

An LTS can be converted into a BA which accepts exactly the same traces:

Lemma 4.1 Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS. $\pi = e_0, e_1, \ldots$ is a trace of L if and only if it is a trace of the BA $B = \langle Q, q_0, A, \Delta, Q \rangle$.

The BA contains the same states and transitions as the LTS and all of its states are accepting. This ensures that, if a trace of the LTS is also a trace of the BA, it will enter at least an accepting state of the BA an infinite number of times, since the set of states is finite. Every infinite trace of the BA is an infinite trace of the LTS by construction. We call LTS2BA the procedure for converting an LTS into a BA. This procedure copies all the states and transitions from the LTS to the BA and adds all the states of the LTS in the set of the accepting states of the BA. For example, the BA obtained from the LTS in Fig. 1a has the same states and transitions, and states 1, 2 and 3 are also accepting.

The *intersection operator* computes the synchronous product of two automata: given two BA M and N, it computes a BA whose traces are the intersection of the traces of M and N.

Definition 4.7 [CGP99] Let $M = \langle Q_M, q_{0,M}, A, \Delta_M, Q_{a,M} \rangle$ and $N = \langle Q_N, q_{0,N}, A, \Delta_N, Q_{a,N} \rangle$ be BA defined over the same alphabet A. Their *intersection* is a symmetric operator ($\cap$) such that $P = M \cap N$ is a BA $P = \langle Q_M \times Q_N \times \{0, 1, 2\}, \langle q_{0,M}, q_{0,N}, 0 \rangle, A, \Delta, Q_{a,M} \times Q_{a,N} \times \{2\} \rangle$, where a transition $(\langle q_m, q_n, x \rangle, A, \langle q'_m, q'_n, y \rangle)$ is in Δ if and only if:

1. $(q_m, e, q'_m) \in \Delta_M$ and $(q_n, e, q'_n) \in \Delta_N$;
2. if $x = 0$ and $q'_m \in Q_{a,M}$ then $y = 1$;
3. if $x = 1$ and $q'_n \in Q_{a,N}$ then $y = 2$;
4. if $x = 2$ then $y = 0$;
5. otherwise, $y = x$.

The states of the system contain all the possible combinations of the states of the automata M and N and the values $\{0, 1, 2\}$. A transition from a state $\langle q_m, q_n, x\rangle$ to a state $\langle q'_m, q'_n, y\rangle$ labeled with e is in the intersection if both M and N can move from q_m (resp. q_n) to q'_m (resp. q'_n) on a transition labeled with e. The third component in each state is responsible for guaranteeing that an infinite trace of the intersection automaton is present if and only if it corresponds to an infinite trace of M (resp. N) in which an accepting state is visited infinitely often. The third component is initially 0, and changes from 0 to 1 and from 1 to 2 whenever an accepting state of M and N is entered, respectively. After an accepting state of N is visited, it is set back to 0. This construction guarantees that every infinite trace of the intersection automaton is an infinite trace of both M and N, i.e., where accepting states of M and N are visited infinitely often.

4.3. Fluent linear time temporal logic

It is often non-trivial to express LTL properties directly in terms of events, especially when the interest is to define intervals among the occurrence of different events and relationships between them. Thus, in this work, we consider FLTL [San95, CDGV02, GM03] as a logic to express properties of the system.

A *fluent* is a property that holds after an event occurs and ceases to hold when it is terminated by another event.

Definition 4.8 [San95] A *fluent* Fl is a tuple $\langle I_{Fl}, T_{Fl}, Init_{Fl}\rangle$ where $I_{Fl} \subset Act$ is the set of *initiating events*, $T_{Fl} \subset Act$ with $I_{Fl} \cap T_{Fl} = \emptyset$ is the set of *terminating events* and $Init_{Fl} \in \{true, false\}$ is the initial valuation.

A fluent may be *true* or *false*. The initial value of the fluent is specified using the attribute $Init_{Fl}$ [MS99]. A fluent is *true* if it has been initialized by an event $e_1 \in I_{Fl}$ at an earlier time point (or if $Init_{Fl} = true$, i.e., it was initially *true*) and has not yet been terminated by another event $e_2 \in T_{Fl}$; otherwise, it is *false*. For example, the fluent $PR = \langle\{prodInfoReq\}, \{prodCancel, prodReq\}, false\rangle$, representing the fact that the furniture-sale component is processing a request, is initially *false*. It becomes *true* from the moment when the event *prodInfoReq* occurs, i.e., a request is received, until the moment in which the event *prodCancel*, which cancels the order request, or the event *prodReq*, which confirms the order request, occur.

Fluents also implicitly allow developers to specify properties on event occurrence. The occurrence of an event $e \in Act$ can be considered by defining a fluent where the initial set of actions is the singleton $\{e\}$ and the terminating set contains all other actions in the alphabet of the system, i.e., $Act \setminus \{e\}$. In the following, we use the notation *F_Event* to indicate a fluent that is *true* when the event with label *event* occurs.

Definition 4.9 Given a set of fluents Φ, *an FLTL formula* ϕ has the following syntax:

$$\phi := Fl \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc\phi \mid \Diamond\phi \mid \Box\phi \mid \phi\,\mathcal{U}\,\phi \mid \phi\,\mathcal{W}\,\phi, \text{ where } Fl \in \Phi$$

$\bigcirc$ (next), $\Diamond$ (eventually), $\Box$ (always), $\mathcal{U}$ (until) and $\mathcal{W}$ (weak until) are the standard LTL operators.

For example, the FLTL property $\Diamond(PR)$ states that eventually the furniture-sale component will process a request. The FLTL encodings of the properties *P1*, *P2*, *P3* and *P4* are shown in Fig. 3a. Note that, for completeness, Definition 4.9 provides the complete syntax of FLTL formulae, including the operators $\Diamond$ (eventually), $\Box$ (always) and $\mathcal{W}$ (weak until), but all the FLTL formulae can be expressed using only the operators $\bigcirc$ and $\mathcal{U}$.

The semantics of FLTL formulae is given by considering infinite sequences of fluents associated with *infinite* traces of an LTS. We begin by constructing an FLTL interpretation of an infinite trace.

Definition 4.10 Let $Act_\tau = Act \cup \{\tau\}$. Given a set of fluents Φ and an infinite trace $\pi = e_0, e_1, \ldots$ over Act_τ, *an* FLTL *interpretation* of π is an infinite sequence $f_0, f_1, \ldots$ over 2^Φ which assigns to each index i of π the set of fluents that hold in position i. Formally, $\forall\, i \in \mathbb{N}, \forall\, Fl \in \Phi$, $Fl \in f_i$ if and only if either of the following conditions hold:

- $Init_{Fl} = true$ and $(\forall\, k \in \mathbb{N},\ k \leq i,\ e_k \notin T_{Fl})$;
- $\exists\, j \in N$ such that $(j \leq i)$ and $(e_j \in I_{Fl})$ and $(\forall\, k \in \mathbb{N}, j < k \leq i,\ e_k \notin T_{Fl})$.

Consider an infinite trace of the LTS in Fig. 1a, in which the events *prodInfoReq*, *infoRcv*, *prodReq* are repeated infinitely often, and a set of fluents $\{PR\}$. The infinite interpretation of this trace is an infinite sequence in which the the set $\{PR\}$ is assigned to each index i where events *prodInfoReq* or *infoRcv* occur and the set $\{\}$ is assigned to each index j where the event *prodReq* occurs. In this sequence, the fluent *PR* holds from the time of occurence of the event *prodInfoReq* until occurence of *prodReq* (that time point itself is excluded).

The FLTL infinite semantics specifies when a formula ϕ is satisfied by the FLTL interpretation of an infinite trace.

Definition 4.11 Let $\pi = e_0, e_1, \ldots$ be an infinite trace, $\rho = f_0, f_1, \ldots$ be its FLTL interpretation and ϕ be an FLTL formula. *The FLTL infinite semantics* of ϕ over ρ is defined as follows:

$$\begin{array}{lll}
\rho_i \models Fl & \Leftrightarrow & Fl \in \rho_i \\
\rho_i \models \neg\phi & \Leftrightarrow & \rho_i \not\models \phi \\
\rho_i \models \phi_1 \wedge \phi_2 & \Leftrightarrow & \rho_i \models \phi_1 \text{ and } \rho_i \models \phi_2 \\
\rho_i \models \bigcirc(\phi) & \Leftrightarrow & \rho_{i+1} \models \phi \\
\rho_i \models (\phi_1)\mathcal{U}(\phi_2) & \Leftrightarrow & \exists\, k \geq i \text{ such that } \rho_k \models \phi_2, \text{ and } \forall\, j \text{ such that } i \leq j < k, \text{ it holds that } \rho_j \models \phi_1,
\end{array}$$

where ρ_i indicates the infinite sub-trace of ρ starting at position i.

The semantics of $\Diamond$ (eventually), $\Box$ (always) and $\mathcal{W}$ (weak until) is not presented since it is standard and can be deduced from the semantics of the operators $\bigcirc$ and $\mathcal{U}$.

Given an infinite trace π and a formula ϕ, we say that π *satisfies* ϕ (denoted $\pi \models \phi$) if and only if the FLTL interpretation ρ of π satisfies ϕ (i.e., $\rho \models \phi$). For example, the infinite trace of the LTS in Fig. 1a in which the events *prodInfoReq*, *infoRcv*, *prodReq* are repeated infinitely often satisfies the FLTL property $\Diamond(PR)$. Note that since FLTL is a natural extension of LTL, it is closed under negation. The proof can be obtained, for example, by applying the procedure described in [YPA06].

4.4. Verifying FLTL properties on LTS

Verification of an FLTL property ϕ on an LTS L aims at checking whether ϕ holds on L. We first define the notion of satisfaction of an FLTL ϕ on an LTS L and then review a procedure for checking an FLTL ϕ on an LTS L.

Definition 4.12 Let L be an LTS and ϕ be an FLTL formula. We say that the LTS L satisfies the FLTL formula ϕ, i.e., $L \models \phi$, if for every infinite trace π of L, it holds that $\pi \models \phi$.

Consider the LTS in Fig. 1a and the fluent PR. The FLTL formula $\Diamond PR$ holds on this LTS.

We now recall a procedure to check an FLTL formula ϕ on an LTS L. Note that the infinite traces of the LTS are defined over the set of events A, while the formula is defined over the set of fluent propositions Φ. This gap has been handled by introducing fluent automata and synchronizer automata. These automata are used in the verification procedure to bind event occurrence with the fluent satisfaction. We first present fluent automata and synchronizer automata and then describe how they are used in the verification procedure.

A *fluent automaton* relates the occurrence of events with the satisfaction of a given fluent.

Definition 4.13 Let Φ be a set of fluents and Fl=$\langle I_{Fl}, T_{Fl}, Init_{Fl}\rangle$ be a fluent in Φ. Its *fluent automaton* is an LTS $F = \langle Q, q_0, A, \Delta\rangle$, where:

- $Q = \{q_t, q_f\}$;
- $A = I_{Fl} \cup T_{Fl} \cup 2^{\Phi}$;
- $\Delta = \{(q_f, e, q_t) \mid e \in I_{Fl}\} \cup$
  $\{(q_t, e, q_t) \mid e \in I_{Fl}\} \cup$
  $\{(q_t, e, q_f) \mid e \in T_{Fl}\} \cup$
  $\{(q_f, e, q_f) \mid e \in T_{Fl}\} \cup$
  $\{(q_f, x, q_f) \mid x \in 2^{\Phi} \text{ and } Fl \notin x\} \cup$
  $\{(q_t, x, q_t) \mid x \in 2^{\Phi} \text{ and } Fl \in x\}$;
- $q_0 = q_t$ if initially $Init_{Fl} = true$, else $q_0 = q_f$.

The automaton has two states: q_t, where the fluent holds, and q_f, where it does not. The automaton moves from q_t (resp. q_f) to q_f (resp. q_t) if an event from its initiating (resp. terminating) set occurs. Fluent satisfaction is represented by self-transitions labeled with the fluent propositions from 2^{Φ}. For example, consider the set of fluents $\Phi = \langle PR, FR\rangle$, where PR=$\langle\{prodInfoReq\}, \{prodCancel, prodReq\}, false\rangle$, and FR is an additional arbitrary fluent. The fluent automaton associated with the fluent PR is represented in Fig. 4.

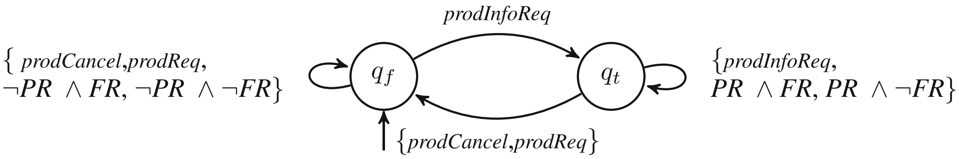


Fig. 4. Fluent automaton associated with the fluent PR

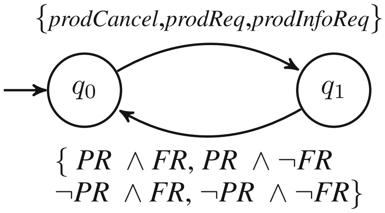


Fig. 5. Synchronizer automaton associated with the fluent PR and FR and the set of events $A = \{prodInfoReq, prodCancel, prodReq\}$

In Fig. 4, multiple transitions with the same source and destination states are represented by a single transition, where the atomic propositions that label the different transitions are separated by commas. For example, the transition labeled with the atomic propositions $\{prodInfoReq, PR \wedge FR, PR \wedge \neg FR\}$ indicates three different transitions, labeled with $prodInfoReq$, $PR \wedge FR$ and $PR \wedge \neg FR$. If a transition labeled with PR is fired, the fluent PR holds, while if a transition labeled with $\neg PR$ is fired, the fluent PR does not hold. As expected, infinite traces of the automaton indicate that PR holds only after an event $prodInfoReq$ and before events $\{prodCancel, prodReq\}$. Furthermore, PR does not hold initially and becomes *false* after occurence of either of the events $prodCancel$, $prodReq$ and before a transition labeled with $prodInfoReq$ is fired.

While the fluent automaton allows taking a transition labeled with a fluent (e.g., PR in Fig. 4) or its negation (e.g., $\neg PR$ in Fig. 4), nothing forces these transitions to actually take place. Instead, we aim to generate an automaton that is forced to alternate the execution of transitions labeled with events and fluents. This automaton is obtained by combining the fluent automaton with a special *synchronizer* automaton, which we define below.

Definition 4.14 Let Φ be a set of fluents and Act be a set of events, such that for all $Fl \in \Phi$, $I_{Fl} \subseteq Act$ and $T_{Fl} \subseteq Act$, the *synchronizer automaton* is an LTS $Sync = \langle Q, q_0, A, \Delta \rangle$, where

- $Q = \{q_0, q_1\}$;
- $A = Act \cup 2^{\Phi}$;
- $\Delta = \{(q_1, Fl, q_0) \mid Fl \in 2^{\Phi}\} \cup \{(q_0, e, q_1) \mid e \in Act\}$.

The synchronizer automaton has two states: q_0 and q_1. It moves from q_0 to q_1 by firing a transition labeled with an event, and from q_1 to q_0 by firing a transition labeled with a fluent. Thus, it alternatively executes transitions labeled by events and by fluents. For example, the synchronizer automaton associated with the fluents PR and FR and the set of events $A = \{prodInfoReq, prodCancel, prodReq\}$ is shown in Fig. 5.

Algorithm 1 Checks an FLTL formula on an LTS.

```
function CHECKFLTLONLTS($L, \phi, F_1, F_2, \ldots F_n, Sync$)
    $B \Leftarrow$ LTL2BA($\neg\phi$)
    $P \Leftarrow (L \parallel F_1 \parallel F_2 \parallel \ldots F_n \parallel Sync)$
    $P' \Leftarrow$ LTS2BA($P$)
    $B' \Leftarrow$ ADDEVENTS($B, Act$)
    $\mathcal{I} \Leftarrow B' \cap P'$
    return CHECKEMPTINESS($\mathcal{I}$)
```

The synchronizer and the fluent automata are used to verify FLTL formulae on LTS via a procedure described in Algorithm 1, in which the classical model checking algorithm presented in [CGP99] is enriched to deal with

FLTL instead of LTL and the combination of LTS and BA. The procedure takes as parameters the LTS L, the formula ϕ, the fluent automata $F_1, F_2, \ldots F_n$, and the synchronizer $Sync$. It returns *true* if the property ϕ is satisfied, and *false* and a counterexample if it is not. The algorithm translates the negation of the formula into a BA B by using a standard LTL2BA procedure [VW94] (Line 2). Then, it computes the parallel composition between the LTS L, the fluent automata $F_1, F_2, \ldots F_n$ and the synchronizer $Sync$ (Line 3). In the traditional automata-based approach, the next step is to compute the intersection between the BA representing the system and the BA representing the negation of the LTL property to check. Since in our case, the model is an LTS, it needs to be converted in a BA via the function LTS2BA (Line 4). Moreover, as the transitions of the obtained BA are labeled with fluents and transitions of the model obtained in Line 3 alternate fluents and events, to correctly compute the intersection between the model and the formula (Line 6), we need to add a self-loop labeled with the events in Act to all the states in B (Line 5). The reason is that the intersection that is computed in the next step of the algorithm (Line 6) synchronously fires transitions of the automata B' and P'. Thus, transitions labeled with events of P' can only fire if the BA obtained from the property also contains transitions labeled with events. Essentially, the BA P' fires its transitions depending on the behavior of the system and on the relation between events and fluents. The intersection extracts from those behaviors the ones that also satisfy the property of interest as specified by the fluent-labeled transitions present in the automaton B'. The function CHECKEMPTINESS is then used to verify whether the automaton $\mathcal{I}$ is empty.

Theorem 4.1 Algorithm 1 returns *true* if and only if there exists an infinite trace of the LTS L which satisfies the FLTL formula.

5. Fluent linear time temporal logic and finite traces

Section 4 described how FLTL formulae are evaluated on infinite traces. However, it might be necessary to evaluate satisfaction of formulae on finite traces, e.g., recall that pre- and post-conditions introduced in Sect. 3 specify properties over traces that reach unspecified components. In this section,

- we define finite semantics of FLTL formulae, referred to as FLTL$_f$ (Sect. 5.1);
- we present a procedure for checking FLTL$_f$ formulae on LTS$_f$ (Sect. 5.2). Our procedure reuses the algorithm for checking the satisfaction of FLTL formulae on LTS presented in Sect. 4.4;
- we present a procedure for taking an FLTL$_f$ formula ϕ and synthesizing from it an LTS$_f$ that exhibits exactly those finite traces that satisfy ϕ (Sect. 5.3). This procedure forms one of the main tools used in FIDDle for providing automatic support for iterative design of components, as described in Sect. 7.

5.1. FLTL$_f$ semantics on finite traces

Interpretation of FLTL formulae on finite traces is denoted by FLTL$_f$. FLTL$_f$ syntax is the same as FLTL and its semantics is inspired by [DGV13].

We begin by defining the interpretation of a fluent Fl on a finite trace.

Definition 5.1 Let Φ be a set of fluents, and $\pi = e_0, e_1, \ldots, e_n$ be a finite trace over Act. *An* FLTL$_f$ *interpretation* of π is a finite sequence $f_0, f_1, \ldots, f_n$ over 2^Φ which assigns to each index i of π the set of fluents that hold in position i for $0 \leq i \leq n$, by following the same rules as specified in Definition 4.10.

Consider the finite trace *shipInfoReq*, *costAndTime*, *prodInfoReq*, *infoRcvd* of the LTS$_f$ in Fig. 3e and the fluent $W = \langle\{$*shipInfoReq*, *prodInfoReq*$\}, \{$*costAndTime*, *infoRcvd*$\}$, *false*$\rangle$, which is *true* when the system is waiting for some information. The fluent W becomes *true* when a request is performed by the component, i.e., a *shipInfoReq* or a *prodInfoReq* event occur, and becomes *false* when some information is received (*costAndTime* or *infoRcvd* occurs). The FLTL$_f$ *interpretation* of the trace *shipInfoReq*, *costAndTime*, *prodInfoReq*, *infoRcvd* is $\{W\}, \{\}, \{W\}, \{\}$.

We now use the interpretation of a finite trace to define the FLTL$_f$ semantics.

Definition 5.2 Let $\pi = e_0, e_1, \ldots, e_n$ be a finite trace, $\rho = f_0, f_1, \ldots, f_n$ be its FLTL$_f$ interpretation, and ϕ be a FLTL$_f$ formula. The FLTL$_f$ finite semantics of ϕ over ρ is as defined by Definition 4.11, with the exception of the operators $\bigcirc$ and $\mathcal{U}$ defined as follows:

$$\rho_i \models \bigcirc(\phi) \quad \Leftrightarrow \quad i < n \wedge \rho_{i+1} \models \phi$$
$$\rho_i \models (\phi_1)\mathcal{U}(\phi_2) \quad \Leftrightarrow \quad \exists\, i \leq k \leq n \text{ such that } \rho_k \models \phi_2 \text{ and } \forall\, i \leq j < k,\ \rho_j \models \phi_1,$$

where ρ_i refers to the finite sub-trace of ρ starting at position i.

Intuitively, a formula preceded by the operator $\bigcirc$ does not hold in the last position of the interpretation, since the operator $\bigcirc$ forces the existence of the next position of the interpretation. A formula of the type $(\phi_1)\mathcal{U}(\phi_2)$ forces the fluent ϕ_2 to occur before reaching the position n of the interpretation. For example, the formula $\bigcirc(W)$, where W is the fluent previously defined, does not hold on the finite trace *shipInfoReq*, *costAndTime*, *prodInfoReq*, *infoRcvd*. Given a finite trace π its FLTL$_f$ interpretation ρ and a formula ϕ, we say that $\pi \models \phi$ if and only if $\rho \models \phi$.

5.2. Verifying FLTL$_f$ properties on LTS$_f$

We begin by defining what it means for an LTS$_f$ L to satisfy an FLTL$_f$ formula, and then present a verification procedure that checks the satisfaction of an FLTL$_f$ formula ϕ on an LTS$_f$ L. We reduce this problem to checking an FLTL formula ϕ' on an LTS L', where ϕ' and L' are obtained from ϕ and L. We begin by showing that checking whether an FLTL$_f$ formula ϕ holds on a finite trace π can be done by evaluating whether an equivalent FLTL formula ϕ' holds on the *infinite extension* $e(\pi)$ of the trace ϕ.

Definition 5.3 Let $\pi = e_0, e_1, \ldots, e_n$ be a finite trace defined over the set of events Act_τ such that *end* $\not\in Act_\tau$. Its *infinite extension*, $e(\pi) = e_0, e_1, \ldots, e_n, \{end\}^\omega$, is obtained by concatenating an infinite number of occurences of the event *end* to the trace π.

Definition 5.4 Let F be an LTS$_f$ and ϕ be an FLTL$_f$ formula. We say that the LTS$_f$ F *satisfies the FLTL$_f$ formula* ϕ, i.e., $F \models \phi$, if for every finite trace π of F, it holds that $\pi \models \phi$.

Consider the LTS$_f$ in Fig. 3e and the fluent W. The FLTL$_f$ formula $\Diamond W$ holds on this LTS$_f$.

Definition 5.5 Let Φ be a set of fluents defined over the alphabet Act_τ, such that *end* $\not\in Act_\tau$, Φ' be a set of fluents defined over the alphabet $Act_\tau \cup \{end\}$, ϕ be an FLTL$_f$ formula defined over Φ, and ϕ' be an FLTL formula defined over Φ'. ϕ' is *equivalent* to ϕ if for every finite trace π , $\pi \models \phi \Leftrightarrow e(\pi) \models \phi'$.

We now want to define an operator FLTL$_f$2FLTL that, given an FLTL$_f$ formula ϕ, generates an equivalent FLTL formula ϕ'.

Definition 5.6 Let Φ be a set of fluents defined over the alphabet Act_τ, such that *end* $\not\in Act_\tau$, *END* be the fluent $\langle\{end\}, Act_\tau, \mathit{false}\rangle$, and ϕ be an FLTL$_f$ formula defined over Φ. The operator FLTL$_f$2FLTL converts ϕ into an FLTL by applying the following rules:

$$\text{FLTL}_f\text{2FLTL}(Fl) \rightarrow Fl', \text{ where } Fl = \langle I_{Fl}, T_{Fl}, Init_{Fl}\rangle \text{ and } Fl' = \langle I_{Fl}, T_{Fl} \cup \{end\}, Init_{Fl}\rangle;$$
$$\text{FLTL}_f\text{2FLTL}(\neg\phi) \rightarrow \neg\text{FLTL}_f\text{2FLTL}(\phi);$$
$$\text{FLTL}_f\text{2FLTL}(\phi_1 \wedge \phi_2) \rightarrow \text{FLTL}_f\text{2FLTL}(\phi_1) \wedge \text{FLTL}_f\text{2FLTL}(\phi_2);$$
$$\text{FLTL}_f\text{2FLTL}(\bigcirc\phi) \rightarrow \bigcirc(\text{FLTL}_f\text{2FLTL}(\phi) \wedge \neg END);$$
$$\text{FLTL}_f\text{2FLTL}(\phi_1\,\mathcal{U}\,\phi_2) \rightarrow \text{FLTL}_f\text{2FLTL}(\phi_1)\mathcal{U}(\text{FLTL}_f\text{2FLTL}(\phi_2) \wedge \neg END).$$

Let $\phi = \bigcirc\neg W$ be an FLTL$_f$ formula, and $\pi = e_0, e_1, \ldots, e_n$ be a finite trace. We discuss the evaluation of the formula in the final position n of the trace. The formula $\bigcirc\neg W$ is *false* in position n by Definition 5.2, as $i < n$ is not satisfied. Let us consider the extension $e(\pi)$ of π. The FLTL$_f$ formula $\bigcirc((\neg W) \wedge \neg END)$ obtained from ϕ by applying the function FLTL$_f$2FLTL is *false* in position n, as intended. This formula evaluates to *true* in position n only after adding the event *end* to the set of the terminating events of the fluent W.

Consider a finite trace π defined over the set of events Act_τ, such that *end* $\not\in Act_\tau$, its infinite extension $e(\pi)$, and an FLTL$_f$ formula ϕ. We show that the problem of checking the satisfaction of ϕ can be reduced to checking its equivalent formula ϕ' on $e(\pi)$.

Theorem 5.1 Let Φ be a set of fluents, where $END \not\in \Phi$, ϕ be an FLTL$_f$ formula defined over the set Φ, and ϕ' be the formula constructed using the FLTL$_f$2FLTL procedure. The FLTL formula ϕ' is equivalent to ϕ.

Proof Sketch We show that $\pi \models \phi \Leftrightarrow e(\pi) \models \phi'$ holds in the base case, when ϕ is a fluent, and that it also holds when temporal operators are considered.

Consider the finite interpretation ρ of $\pi = e_0, e_1, \ldots, e_n$ and the infinite interpretation ρ' of $e(\pi)$. For every $Fl \in \Phi$ and position i such that $i \leq n$, where n is the length of the finite trace, $\rho_i \models Fl \Leftrightarrow \rho'_i \models Fl'$ by construction. Furthermore, for every $Fl \in \Phi$ and position i such that $i > n$, $\rho_i \not\models Fl$ and $\rho'_i \not\models Fl'$.

For operators $\neg$ and $\wedge$, $\rho \models \neg\phi \Leftrightarrow \rho'_i \models \neg\text{FLTL}_f\text{2FLTL}(\phi)$ and $\rho \models \phi_1 \wedge \phi_2 \Leftrightarrow \rho'_i \models \text{FLTL}_f\text{2FLTL}(\phi_1) \wedge \text{FLTL}_f\text{2FLTL}(\phi_2)$ by construction.

Consider the operator $\bigcirc$. If $i < n$, $\rho_i \models \bigcirc\phi$ if and only if $\rho'_i \models \bigcirc(\text{FLTL}_f\text{2FLTL}(\phi))$, since *END* is *false* if $i < n$. If $i \geq n$, then, by Definition 5.2, $\rho_i \not\models \bigcirc\phi$, but also $\rho'_i \models \bigcirc \mathit{false}$ by Definition 5.6 and the fact that *END* is *true*.

Consider the operator $\mathcal{U}$. If $i < n$, $\rho_i \models \phi_1 \,\mathcal{U}\, \phi_2$ if and only if $\rho'_i \models \text{FLTL}_f\text{2FLTL}(\phi_1)\mathcal{U}(\text{FLTL}_f\text{2FLTL}(\phi_2))$ by Definition 5.6 and the fact that *END* is *false*. If $i \geq n$, then, by Definition 5.2, $\rho_i \not\models \phi_1 \,\mathcal{U}\, \phi_2$. However, $\rho'_i \models \text{FLTL}_f\text{2FLTL}(\phi_1)\mathcal{U} \mathit{false}$, is also always *false*, since *END* is *true*. □

Consider the finite trace *shipInfoReq*, *costAndTime*, *prodInfoReq*, *infoRcvd*, of the LTS_f shown in Fig. 3e and its infinite extension *shipInfoReq*, *costAndTime*, *prodInfoReq*, *infoRcvd*, end^{ω}. The satisfaction of the FLTL_f formula $\phi = \bigcirc(W)$, where $W = \langle\{\mathit{shipInfoReq}, \mathit{prodInfoReq}\}, \{\mathit{costAndTime}, \mathit{infoRcvd}\}, \mathit{false}\rangle$, on the finite trace can now be checked by evaluating the satisfaction of the formula $\phi = \bigcirc(W \wedge \neg \mathit{END})$ on its infinite extension.

We show that checking satisfaction of an FLTL_f formula on LTS_f can be reduced to checking the corresponding FLTL formula on an LTS.

Theorem 5.2 Let ϕ be an FLTL_f formula and $F = \langle Q, q_0, A, \Delta, Q_f\rangle$ be an LTS_f model. If ϕ' is the FLTL formula equivalent to ϕ and $F' = \langle Q, q_0, A \cup \{\mathit{end}\}, \Delta'\rangle$ is an LTS such that $\Delta' = \Delta \cup \{(q, \mathit{end}, q) \mid q \in Q_f\}$, then $F \models \phi \Leftrightarrow F' \models \phi'$.

Intuitively, F' is replaced by adding a self-loop labeled with the event *end* to all of the final states of the LTS_f. Consider for example the LTS_f presented in Fig. 3e. The FLTL_f formula $\phi = \bigcirc(W)$ can be checked by verifying that the formula $\phi = \bigcirc(W \wedge \neg \mathit{END})$ holds on the LTS obtained by an *end* self-loop to the state 2.5. In the rest of this paper, we refer to this procedure as $\text{CHECKFLTL}_f\text{ONLTS}_f$.

We now provide the proof of Theorem 5.2.

Proof Sketch The infinite traces of the LTS F' are the infinite extensions of the finite traces of F; thus, by Theorem 5.1, the proposed procedure is correct. □

5.3. Synthesizing LTS_f from FLTL_f formulae

Given an FLTL_f formula ϕ, we show how to construct an LTS_f such that its finite traces are exactly those that satisfy formula ϕ.

Definition 5.7 We define an FLTL *end formula* ϕ_{end} as $\Diamond(\mathit{END}) \wedge \Box(\mathit{END} \rightarrow \bigcirc(\mathit{END}))$.

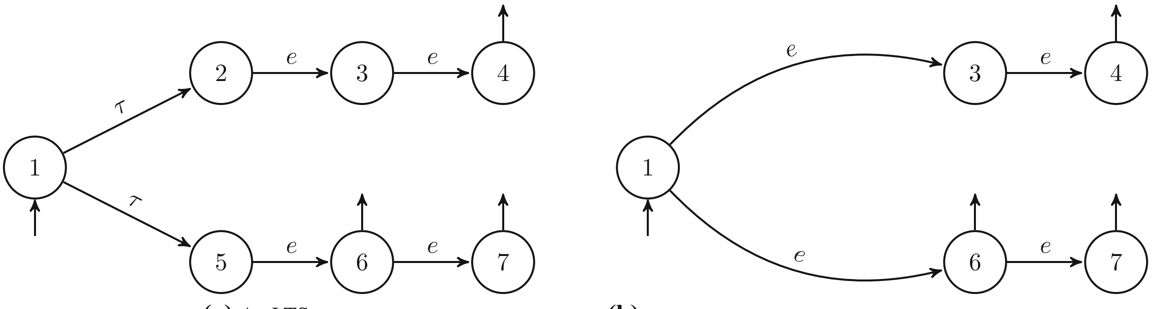
Lemma 5.1 Let π be an infinite trace and ϕ_{end} be an FLTL formula defined via Definition 5.7. If $\pi \models \phi_{\mathit{end}}$, then π has the form $e_0, e_1, \ldots, e_n, \{\mathit{end}\}^{\omega}$.

Proof Sketch The subformula $\Diamond(\mathit{END})$ forces the *END* fluent to eventually hold. The subformula $\Box(\mathit{END} \rightarrow \bigcirc(\mathit{END}))$ specifies that globally if the *END* fluent holds, it must hold in the next position. Since the set of the initiating events of the *END* fluent only contains the event *end* and the set of the terminating events is Act_{τ}, event *end* must occur at some position $n + 1$, and in any position $i \geq n + 1$. □

We also define an operator that converts a BA into an LTS_f and a procedure that removes τ-labeled transitions from a LTS_f.

Definition 5.8 Let $B = \langle Q, q_0, A, \Delta, Q_a\rangle$ be BA. The LTS_f $F = \langle Q, q_0, A, \Delta, Q_f\rangle$ is the *conversion* of the BA B, denoted $F = \text{BA2LTS}_f(B)$, if $Q_f = Q_a$.

Definition 5.9 Let $F = \langle Q, q_0, A, \Delta, Q_f\rangle$ be an LTS_f. $F' = \text{REMOVE}_{\tau}(F)$ is an LTS_f $\langle Q, q_0, A, \Delta', Q_f\rangle$, such that any transition $(q_i, e, q_j) \in \Delta'$ satisfies one of the following conditions: (i) $e \neq \tau$; or (ii) $(q_i, \tau, q_{i+1}), (q_{i+1}, \tau, q_{i+2}), \ldots (q_{j-2}, \tau, q_{j-1}), (q_{j-1}, e, q_j)$ and $e \neq \tau$.



(a) An LTS$_f$.

(b) LTS$_f$ obtained by applying Remove$_\tau$ function on the LTS$_f$ in Fig. 6a.

Fig. 6. An example usage of the procedure Remove$_\tau$

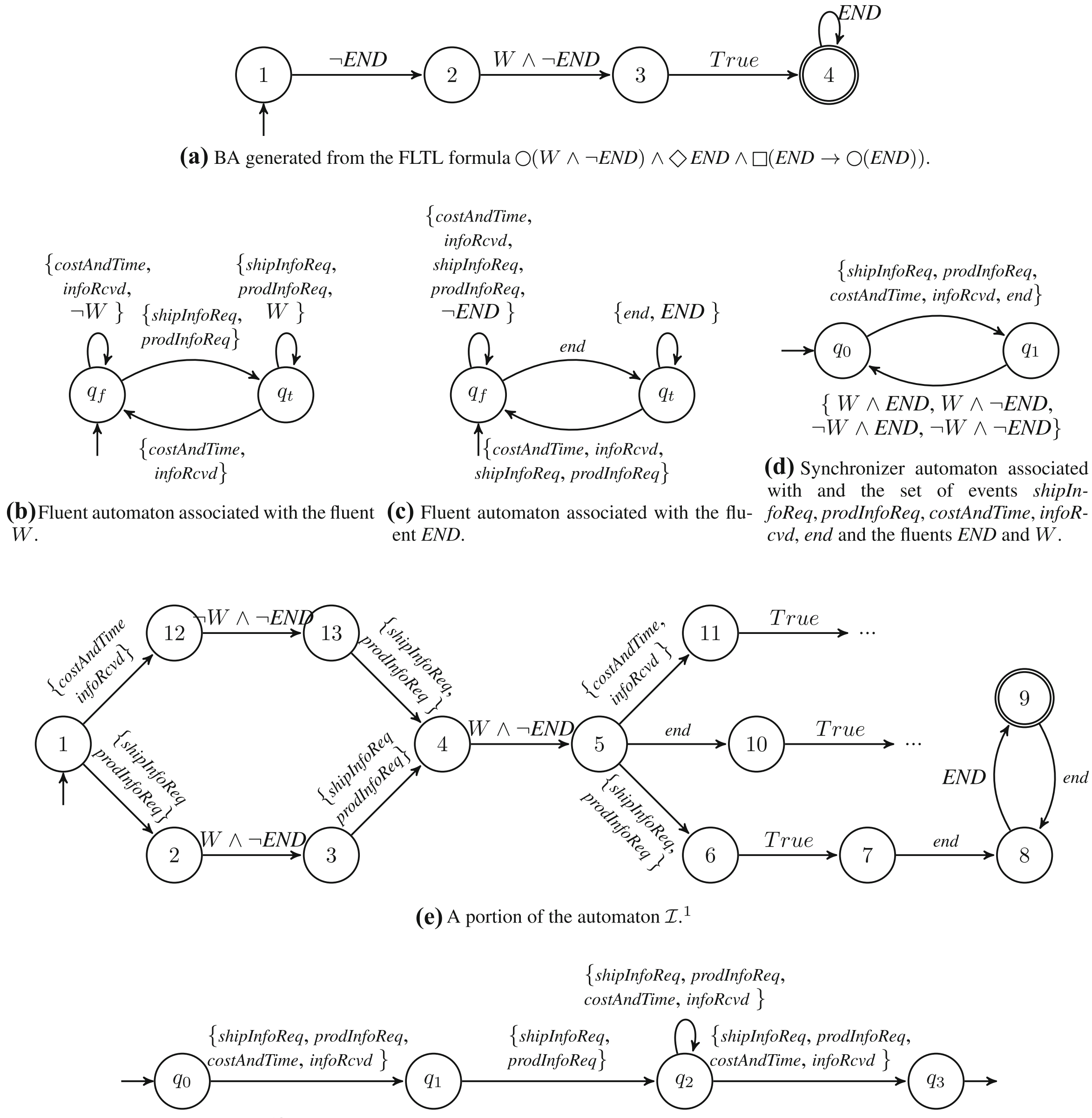
In essence, sequences of τ-labeled transitions that connect a state q_i to a state q_{j-1} followed by a transition to a state q_j labeled with some symbol $e \neq \tau$ are replaced by a transition from q_i to q_j labeled with e. Consider, for example, the LTS in Fig. 6a. The procedure Remove$_\tau$ generates the LTS in Fig. 6b. The transitions $(1, \tau, 2)$, $(2, e, 3)$ are replaced by the single transition $(1, e, 3)$, and $(1, \tau, 5)$, $(5, e, 6)$ are replaced by $(1, e, 6)$.

Algorithm 2 describes a procedure for computing an LTS$_f$ model F whose finite traces satisfy the FLTL$_f$ formula ϕ. We discuss the steps of our algorithm using an example FLTL$_f$ formula $\phi = \bigcirc(W)$, where $W = \langle\{\mathit{shipInfoReq}, \mathit{prodInfoReq}\}, \{\mathit{costAndTime}, \mathit{infoRcvd}\}, \mathit{false}\rangle$. The algorithm begins by transforming ϕ to FLTL via the procedure FLTL$_f$2FLTL (Definition 5.6). For example, the FLTL$_f$ formula $\phi = \bigcirc(W)$ is transformed into the FLTL formula $\phi' = \bigcirc(W \wedge \neg \mathit{END})$. The algorithm then constructs a BA by conjoining the FLTL formula and the end formula ϕ_{end} and applying the procedure LTL2BA (Line 3). In our example, the BA in Fig. 7a is computed from the FLTL formula $\bigcirc(W \wedge \neg \mathit{END}) \wedge \Diamond \mathit{END} \wedge \Box(\mathit{END} \rightarrow \bigcirc(\mathit{END}))$. Then the algorithm computes the parallel composition of the BA obtained from the formula $\phi' \wedge \phi_{\mathit{end}}$ and the BA obtained by applying the function LTS2BA to the parallel composition of the fluent and the synchronizer automata (Line 4), resulting in an automaton whose infinite traces satisfy the FLTL$_f$ formula, and interleaves the occurence of the fluents and the events. In our example, the fluent automata associated with the fluents W and END are shown in Figs. 7b and 7c, and the synchronizer is shown in Fig. 7d. A portion of the intersection between their parallel composition and the BA obtained from the FLTL formula $\phi' \wedge \phi_{\mathit{end}}$ is shown in Fig. 7e. Then, Algorithm 2 converts the parallel composition into an LTS (see Definition 5.8) and hides transitions labeled with fluents and the event *end*. In our example, the transitions from 2 to 3, from 4 to 5, from 6 to 7, from 7 to 8, from 8 to 9, from 9 to 8 and from 12 to 13 in Fig. 7e are hidden, i.e., they are relabeled using the τ symbol. The final step, on Line 7, removes τ-transitions and returns. In our example, the LTS$_f$ for the FLTL formula $\phi = \bigcirc(W)$ is shown in Fig. 7f.

Theorem 5.3 Let ϕ be an FLTL$_f$ formula. Algorithm 2 generates an LTS$_f$ such that its finite traces are exactly those that satisfy the formula ϕ.

Proof Sketch We prove the correctness by construction. Line 3 forces each infinite trace of A to finish with an infinite sequence of END fluents (Lemma 5.1). Thus, the BA reaches an accepting state from which only transitions labeled with the END fluent can be fired. By Theorem 5.1, ϕ' is equivalent to ϕ, so for every infinite trace extension $e(\pi)$, if $\pi \models \phi \Leftrightarrow e(\pi) \models \phi'$. Thus, the BA generated in Line 3 contains infinite traces that are extensions of finite traces π that satisfy the formula ϕ. Line 4 computes an automaton that interleaves fluents and events and ensures that the traces generated by the sequences of events satisfy the property ϕ, as described in Sect. 4.4. Then, the BA is converted into an LTS$_f$ (Line 5). Transitions labeled with fluents and the event *end* are hidden (Line 6). This procedure ensures, by construction, that the traces reaching the final states of the returned LTS$_f$ are exactly those traces that satisfy the formula ϕ. □

[1] For simplicity of presentation, we presented only states 8 and 9, but Algorithm 4.7 generates three copies of these states, for labels {0, 1, 2}.


1
2
3
4
$\neg END$
$W \wedge \neg END$
$True$
END
$\{costAndTime, infoRcvd, \neg W\}$
$\{shipInfoReq, prodInfoReq, W\}$
$\{shipInfoReq, prodInfoReq\}$
$\{costAndTime, infoRcvd\}$
q_f
q_t
$\{costAndTime, infoRcvd, shipInfoReq, prodInfoReq, \neg END\}$
$\{end, END\}$
end
$\{costAndTime, infoRcvd, shipInfoReq, prodInfoReq\}$
$\{shipInfoReq, prodInfoReq, costAndTime, infoRcvd, end\}$
q_0
q_1
$\{W \wedge END, W \wedge \neg END, \neg W \wedge END, \neg W \wedge \neg END\}$
1
12
13
2
3
4
5
6
7
8
9
10
11
$\{costAndTime, infoRcvd\}$
$\neg W \wedge \neg END$
$\{shipInfoReq, prodInfoReq\}$
$\{shipInfoReq, prodInfoReq\}$
$W \wedge \neg END$
$\{shipInfoReq, prodInfoReq\}$
$W \wedge \neg END$
$\{costAndTime, infoRcvd\}$
$True$
...
end
$True$
...
$\{shipInfoReq, prodInfoReq\}$
$True$
end
END
end
q_0
q_1
q_2
q_3
$\{shipInfoReq, prodInfoReq, costAndTime, infoRcvd\}$
$\{shipInfoReq, prodInfoReq\}$
$\{shipInfoReq, prodInfoReq, costAndTime, infoRcvd\}$
$\{shipInfoReq, prodInfoReq, costAndTime, infoRcvd\}$

(a) BA generated from the FLTL formula $\bigcirc(W \wedge \neg END) \wedge \Diamond END \wedge \Box(END \rightarrow \bigcirc(END))$.

(b) Fluent automaton associated with the fluent W.

(c) Fluent automaton associated with the fluent *END*.

(d) Synchronizer automaton associated with and the set of events *shipInfoReq*, *prodInfoReq*, *costAndTime*, *infoRcvd*, *end* and the fluents *END* and W.

(e) A portion of the automaton $\mathcal{I}$.[1]

(f) LTS$_f$ for the FLTL formula $\phi = \bigcirc(W)$ computed using Algorithm 2.

Fig. 7. Application of the steps of Algorithm 2 over the formula $\bigcirc(W)$

Algorithm 2 Transforms an FLTL$_f$ formula into an equivalent LTS$_f$.

```
function FLTL_f2 LTS_f($\phi$, $Fl_1$, $Fl_2$, $Fl_3$, $\ldots$, $Fl_n$, $Sync$)
    $\phi' \Leftarrow$ FLTL_f2FLTL($\phi$)
    $A \Leftarrow$ LTL2BA($\phi' \wedge \phi_{end}$)
    $B \Leftarrow A \cap$ LTS2BA($Fl_1 \parallel Fl_2 \parallel Fl_3 \ldots \parallel Fl_n \parallel Sync$)
    $C \Leftarrow$ BA2LTS_f($B$)
    $D \Leftarrow C \backslash (\Phi \cup \{END, end\})$
    return REMOVE_τ($D$)
```

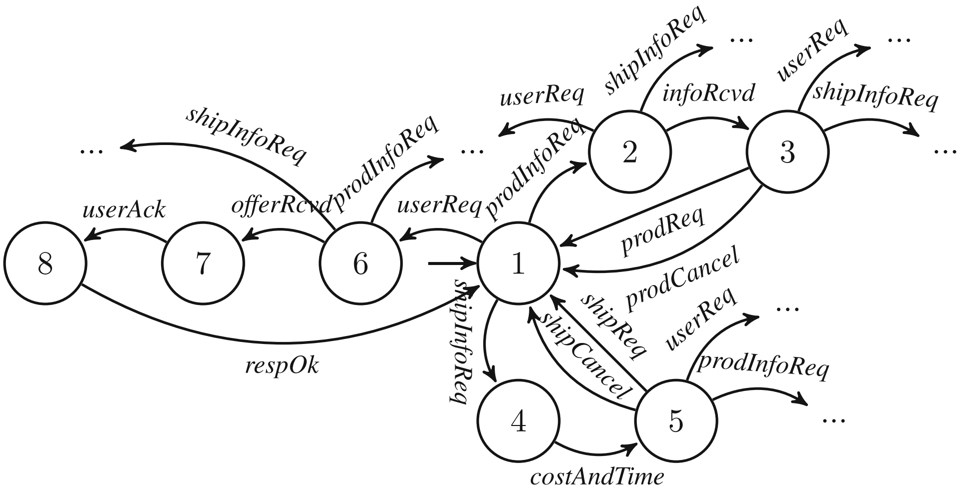


Fig. 8. The environment $\mathcal{E}$ obtained by computing the parallel composition of the LTS in Figs. 1a, 1b and 1c

6. Modeling notation

This section presents the modeling notation proposed in FIDDle to support the modular, distributed, and iterative development of software controllers. Section 6.1 describes how to model the environment in which the components under development will eventually be deployed. Section 6.2 describes how to provide a high-level model of the components. Section 6.3 describes how pre- and post-conditions can be added to the high level model to facilitate distributed development. Section 6.4 describes how unspecified components can be refined into sub-components by exploiting their pre- and post-conditions.

In the rest of this section, we capitalize generic entities specified in a given modeling formalism and denote specific software development entities using caligraphic fonts. For example, a generic LTS can be indicated by L, while the LTS modeling the behavior of the environment is denoted by $\mathcal{E}$.

6.1. Modeling the environment

The development workflow proposed in FIDDle (Sect. 3) assumes that a model of the environment is initially designed by developers (or provided by a third party) and is not changed during the iterative refinement rounds. FIDDle receives the model of the environment specified through an LTS. The LTS of the environment ($\mathcal{E}$) can be obtained by composing several LTS using the parallel composition operator (Definition 4.4). Each of these LTS models is a component of the environment. For example, the LTS of the environment of the p&d example is obtained via the parallel composition of the LTS described in Figs. 1a, 1b and 1c. The complete model of the environment has 45 states and 174 transitions. A portion of the parallel composition is shown in Fig. 8. Since the alphabets of the three LTS are disjoint, all the transitions are interleaved.

6.2. Initial component design

We propose *partial (finite)* LTS (PLTS) as a formalism to support the high level design of a component under development. A *Partial (finite)* Labeled Transition System is an LTS (resp. LTS$_f$) where some states are "regular" and others are "black-box". Black-box states model portions of the component whose behavior still has to be specified. Each black-box state is augmented with an interface that specifies the universe of events that can occur in the black-box. Partial LTS are used to design an initial component model, partial finite LTS–to define sub-components (see Sect. 6.4).

Definition 6.1 Let $F = \langle Q, q_0, A, \Delta, Q_f \rangle$ be an LTS$_f$. A *Partial Finite Labeled Transition System* (PLTS$_f$) is a structure $P = \langle F, R, B, \sigma \rangle$, where:

- R is the set of *regular* states;
- B is the set of *black-box* states;
- $Q = R \cup B$ and $R \cap B = \emptyset$;
- $\sigma : B \rightarrow 2^A$ is the *interface*.

Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS. A *Partial Labeled Transition System* (PLTS) is a structure $P = \langle L, R, B, \sigma \rangle$ defined as above.

LTS (resp. LTS$_f$) are PLTS (resp. PLTS$_f$), where the set of black-box states is empty. For example, the initial design of a component that interacts with the furniture-sale, the shipping service and the user is represented in Fig. 3d. The interfaces of its black-box states are documented in Fig. 3c. The component is defined over the regular states 1 and 3, and the black-box states 2, 4 and 5. When a user request is received (event *userReq*), the black-box state 2 is entered. This black-box state represents an unspecified functionality for computing an offer to the user. The interface specifies that events *prodInfoReq*, *infoRcvd*, *shipInfoReq* and *costAndTime* can occur while the component is in the black-box state 2. The system exits this black-box state via a transition labeled with the event *offerRcvd*. After leaving the unspecified component, the system enters one of the black-box states: 4, if the user accepts the offer (event *userAck*) or 5, if it refuses it (event *userNack*). In turn, these black-box states represent unspecified components that are executed in response to a user request. The interfaces of the black-box states 4 and 5 specify that the events *prodReq*,*shipReq* and *prodCancel*, *shipCancel* can occur while the component is in the black-box states 4 and 5, respectively.

Definition 6.2 Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS, $P = \langle L, R, B, \sigma \rangle$ be a PLTS, and $D = \langle Q_D, q_{0,D}, A_D, \Delta_D \rangle$ be an LTS. The *parallel composition* $P \parallel D$ is an LTS $S = \langle Q \times Q_D, \langle q_0, q_{0,D} \rangle, A \cup A_D, \Delta_S \rangle$, where Δ_S is defined as follows:

(i) $\dfrac{(q, e, q') \in \Delta}{(\langle q, t \rangle, e, \langle q', t \rangle) \in \Delta_S}$, and $e \in A \setminus A_D$ or $e = \tau$;

(ii) $\dfrac{(t, e, t') \in \Delta_D}{(\langle q, t \rangle, e, \langle q, t' \rangle) \in \Delta_S}$, and one of the following is satisfied:

(a) $e \in A_D \setminus A$;

(b) $e = \tau$;

(c) $q \in B$ and $e \in \sigma(q)$.

(iii) $\dfrac{(q, e, q') \in \Delta, (t, e, t') \in \Delta_D}{(\langle q, t \rangle, e, \langle q', t' \rangle) \in \Delta_S}$ and $e \in A \cap A_D$, $e \neq \tau$.

Rule (i) specifies that if the PLTS P fires the transition (q, e, q') of Δ and the event e does not belong to the set of events of A_D or it is τ, P moves to q' while the LTS D does not change its state. Rule (ii) states that the LTS D performs the transition (t, e, t') while the PLTS P remains in state q if one of the following conditions hold: (a) the event e is not an event of P, (b) the event e is the τ event, (c) the PLTS is in a black-box state q and the event e belongs to the interface of q. Rule (iii) specifies that the PLTS P and the LTS D fire the transitions (q, e, q') and (t, e, t') in parallel if the event e is in the set of events of both P and D.

Consider the initial partial component design $\mathcal{C}$ modeled using the PLTS in Fig 3b and the interfaces of its black-box states documented in Fig. 3c. The parallel composition operator allows analyzing how $\mathcal{C}$ behaves when it is executed in its environment $\mathcal{E}$, which, in turn, is an LTS produced by composing the LTS in Figs. 1a, 1b and 1c.

6.3. Adding pre- and post-conditions

A *pre-condition* is an assumption about the history of the system at the point when a given unspecified component is entered. A *post-condition* is a guarantee that is expected to be established by a given unspecified component. A *contract* of an unspecified component, represented as a black-box state in our formalism, consists of the black-box state interface and its pre- and post-conditions.

Definition 6.3 Let Φ_f be the universal set of FLTL$_f$ formulae. An *Interface PLTS* (IPLTS) I is a tuple $\langle P, pre, post \rangle$, where $P = \langle L, R, B, \sigma \rangle$ is a PLTS, $pre : B \rightarrow \Phi_f$ and $post : B \rightarrow \Phi_f$. An *interface finite PLTS* (IPLTS$_f$) is an IPLTS, where P is a PLTS$_f$.

Pre- and post-conditions are defined using FLTL$_f$ formulae, i.e., formulae over finite traces. Intuitively, a pre-condition $pre(b)$ for a black-box state b must hold on any finite trace that reaches b. For example, the pre-condition of the state 4 of the PLTS of Fig. 3d, shown in Fig. 3c, states that globally, on the finite traces that reach state 4, if a user request has been detected, information from the user has been received. The post-condition of the black-box state 4 indicates that the sub-component that has to replace this state has to ensure that a product is requested from the furniture sale component and a shipping request is sent.

To effectively support developers in iterative design, we define what it means for a pre- and a post-condition to be satisfied by the current partial component and when a property of interest is satisfied by the current partial component. Pre-conditions specify properties that hold on finite traces of the system S that reach the black-box state. Since the environment is fixed and does not change during the component development, semantics of pre-conditions for a black-box state b is defined by considering the parallel composition between the partial component under development $\mathcal{C}$ and its environment $\mathcal{E}$. To reach this goal, we first define the notion of a finite trace of the parallel composition among a PLTS P and an LTS D that reaches a state q_d of the PLTS, where the PLTS P represents the partial component P and the LTS D represents its environment $\mathcal{E}$.

Definition 6.4 Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS, $q_d \in Q$ and $P = \langle L, R, B, \sigma \rangle$ be a PLTS. Let $D = \langle Q_D, q_{0,D}, A_D, \Delta_D \rangle$ be an LTS and $S = P \parallel D$ be an LTS $\langle Q_S, q_{0,S}, A_S, \Delta_S \rangle$. We say that a finite trace $e_0, e_1, \ldots, e_n$ of S *reaches* q_d if there exists a finite sequence $\langle q_0, t_0 \rangle, e_0, \langle q_1, t_1 \rangle, \ldots, e_n, \langle q_d, t_{n+1} \rangle$, where for every $0 \leq i \leq n$, we have $(\langle q_i, t_i \rangle, e_i, \langle q_{i+1}, t_{i+1} \rangle) \in \Delta_S$.

For example, considering the PLTS in Fig. 3d and the LTS in Fig. 1c, the finite trace obtained by performing a *userReq* event reaches the black-box state 2 of the PLTS.

We now show how to transform the LTS $S = \mathcal{C} \parallel \mathcal{E}$ into an LTS$_f$ S', denoted by LTS2LTS$_f(\mathcal{C}, q_d, L)$, so that S' contains exactly the finite traces that reach the state q_d.

Definition 6.5 Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS, $q_d \in Q$, and $P = \langle L, R, B, \sigma \rangle$ be a PLTS, $D = \langle Q_D, q_{0,D}, A_D, \Delta_D \rangle$ be an LTS and $S = P \parallel D$ be an LTS $\langle Q_S, q_{0,S}, A_S, \Delta_S \rangle$. The LTS$_f$ F is *a cast* of the PLTS P over D w.r.t. the state q_d, i.e., $F =$ LTS2LTS$_f(P, q_d, D)$, if $F = \langle Q_S, q_{0,S}, A_S, \Delta_S, \{q_d\} \times Q_D \rangle$.

Intuitively, the LTS$_f$ F is obtained from the LTS $S = P \parallel D$ by setting every state obtained by combining a state q_d of P and one from the set Q_D of the states of D and making it a final state of F.

Lemma 6.1 Let L, q_d, P, D and S be the same as in Definition 6.5. The finite traces of LTS2LTS$_f(P, q_d, D)$ are exactly finite traces that reach q_d (see Definition 6.4).

Proof Sketch By construction, a finite sequence $\langle q_0, t_0 \rangle, e_0, \langle q_1, t_1 \rangle, \ldots, e_n, \langle q_d, t_{n+1} \rangle$ exists in the LTS$_f$ F if and only if a finite sequence $\langle q_0, t_0 \rangle, e_0, \langle q_1, t_1 \rangle, \ldots, e_n, \langle q_d, t_{n+1} \rangle$, where for every $0 \leq i \leq n$, $(\langle q_i, t_i \rangle, e_i, \langle q_{i+1}, t_{i+1} \rangle) \in \Delta_S$, exists in S. □

Post-conditions specify which properties should be ensured by black-box states. To give their semantics, we begin by definining what it means to have traces of the system inside a black-box state.

Definition 6.6 Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS, $P = \langle L, R, B, \sigma \rangle$ be a PLTS with $b \in B$, and D be an LTS. Let $S = P \parallel D$ be an LTS, $\pi = e_0, e_1, \ldots$ be an infinite trace of S, and $\langle q_0, t_0 \rangle, e_0, \langle q_1, t_1 \rangle, e_2, \langle q_2, t_2 \rangle, \ldots$ an infinite execution of S. We say that a sub-trace $e_i, e_{i+1}, \ldots, e_k$ of π is *inside* the black-box state b if the sub-sequence $\langle q_i, t_i \rangle, e_i, \langle q_{i+1}, t_{i+1} \rangle, \ldots, \langle q_{k+1}, t_{k+1} \rangle$ is a maximal sub-sequence such that $q_i = q_{i+1} = \ldots = q_{k+1} = b$ and $e_i, e_{i+1}, \ldots, e_k \in \sigma(b)$.

Note that a sub-trace inside a black-box state is *finite*.

Let $\mathcal{C}$ be the p&d supervisor design (Fig. 3d) and $\mathcal{E}$ be the p&d environment (Figs. 1a, 1b and 1c). Let us consider the infinite trace of $\mathcal{C} \parallel \mathcal{E}$ in which the sequence of events *userReq*, *shipInfoReq*,*costAndTimeofferRcvd*,*usrAck*, *shipReq*,*respOk* is repeated an infinite number of times, the sub-trace *shipInfoReq*,*costAndTime* is inside the black-box state 2.

Let $\mathcal{C}$ be a partial component described using an IPLTS $I = \langle P, pre, post \rangle$, and $\mathcal{E}$ be its environment, described using an LTS D. Traces of $P \parallel D$ are *valid* if they satisfy both the pre- and the post-conditions of all the black-box states.

Definition 6.7 Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS, $P = \langle L, R, B, \sigma \rangle$ be a PLTS, $I = \langle P, pre, post \rangle$ be an IPLTS. Let D be an LTS, and S be $P \parallel D$. A trace π of S *satisfies the pre-conditions* of a black-box state $b \in B$ if for every subtrace π_1 of π that reaches a black-box state b, $\pi_1 \models pre(b)$. A trace π of S *satisfies the post-conditions* of a black-box state $b \in B$ if for every sub-trace π_2 inside a black-box state b, $\pi_2 \models post(b)$.

In the proposed definition, a pre-condition specifies a property that must hold in a sub-trace that reaches a black-box state, while a post-condition specifies a property that must hold in a sub-trace performed inside the black-box state. Intuitively, a post-condition is defined as a constraint on the behavior of the system that must hold *while* the system is in the back-block state. An alternative semantics may define a post-condition as a property that must hold when the system exits the black-box state, i.e., a property that must hold in the concatination of the sub-trace that reaches the black-box state and the sub-trace obtained inside the black-box state (the "whole sub-trace"). Our semantic choice was geared towards fitting within an iterative and incremental development process while allowing distributed development of components. We believe that when a development team has to refine a black-box state, it prefers having a condition that constrains the behavior of the component under development, rather than a condition that constrains the composition of the component that has to be developed and all the components that have been executed before entering that component.

Consider the pre-condition of the black-box state 4 of the partial component $\mathcal{C}$ represented by the IPLTS in Fig. 3d. It requires that any finite trace of $\mathcal{C} \parallel \mathcal{E}$, where $\mathcal{E}$ is its environment, that reaches black-box state 4 is such that if a user requests information about a product, the system provides it. The *post-condition* for the black-box state 4 constrains the behavior of $\mathcal{C} \parallel \mathcal{E}$ when the partial component $\mathcal{C}$ is inside the black-box state 4. For our example, it ensures that a product and a shipping request are performed by the furniture-sale service by $\mathcal{C} \parallel \mathcal{E}$ while the partial component is inside the black-box state 4.

Let $\mathcal{C}$ be a partial component defined using an IPLTS and $\mathcal{E}$ be its environment. We say that $\mathcal{C}$ is *well-formed* when all pre- and post-conditions of its black-box states are satisfied by $\mathcal{C} \parallel \mathcal{E}$.

Definition 6.8 Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS, $P = \langle L, R, B, \sigma \rangle$ be a PLTS, $I = \langle P, pre, post \rangle$ be an IPLTS. Let D be an LTS, and $S = P \parallel D$. The IPLTS I is *well-formed* (over D) if for every trace π of S holds that, if π satisfies the post-conditions, then it also satisfies the pre-conditions.

Intuitively, Definition 6.8 specifies when pre- and post-conditions are consistent with each other. It ensures that the pre-conditions are satisfied if the post-conditions are. Consider the pre-condition (see Fig. 3c) of the black-box state 4 of the p&d supervisor design (Fig. 3d), which requires that if a user request is received, the system finally had sent information to the user. This pre-condition is satisfied if the post-condition of the black-box state 2 ensuring that information is sent to the user is also satisfied (Fig. 3c).

We now define what it means for a property ϕ to hold on a well-formed IPLTS.

Definition 6.9 Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS, $P = \langle L, R, B, \sigma \rangle$ be a PLTS, $I = \langle P, pre, post \rangle$ be an IPLTS. Let D be an LTS, and $S = P \parallel D$. We say that the IPLTS I *satisfies* an FLTL property ϕ if it is well-formed (over D) and for every trace π of S that satisfies the post-conditions, $\pi \models \phi$.

In the p&d example, the post-condition $\Diamond(F\_ProdReq) \wedge \Diamond(F\_ShipReq)$ of the black-box state 4 ensures that the parallel composition of the component in Fig. 3d and its environment satisfies property *P3* (Fig. 1d).

6.4. Sub-components and their integration

In this section, we describe how black-box states representing unspecified components can be refined into sub-components and define an integration operator that replaces the black-box states of a partial component with the corresponding sub-components.

We first define the notion of a sub-component.

Definition 6.10 Let $L = \langle Q, q_0, A, \Delta \rangle$ be an LTS, $P = \langle L, R, B, \sigma \rangle$ be a PLTS, $I = \langle P, pre, post \rangle$ be an IPLTS and $b \in B$. A *sub-component for* b is an IPLTS$_f$ $I_f = \langle P', pre', post' \rangle$, where $P' = \langle F, R', B', \sigma' \rangle$ and F is an LTS$_f$ $\langle Q', q_0', \sigma(b), \Delta', Q_f' \rangle$.

A sub-component $\mathcal{R}$ is represented as an IPLTS$_f$ such that its computation starts in its initial state and ends in one of its final states and is defined over a finite LTS with events in the set $\sigma(b)$.

When the design of a sub-component is finished, it is integrated into the initial partial component design. Let $\mathcal{C}$ be a partial component modeled as an IPLTS and $\mathcal{R}$ be a sub-component (modeled as an IPLTS$_f$) for a black-box state b of $\mathcal{C}$. The *integration operator* replaces b with $\mathcal{R}$.

Recall that given a state q (of an LTS or an LTS$_f$), $\Delta(q)^-$ and $\Delta(q)^+$ denote its incoming and outgoing transitions, respectively, and $\Delta(q)$ denotes the union of the incoming and outgoing transitions of q.

Definition 6.11 Let $I = \langle P, pre, post \rangle$ be an IPLTS, where $P = \langle L, R, B, \sigma \rangle$, $L = \langle Q, q_0, A, \Delta \rangle$ and consider one of its black-box states b. Let $I_f = \langle P_f, pre_f, post_f \rangle$ be an IPLTS$_f$, where $P_f = \langle L_f, R_f, B_f, \sigma_f \rangle$ and $L_f = \langle Q_f, q_{0,f}, A_f, \Delta_f, Q_{f,f} \rangle$. The *integration* $I' = \textsc{Integrate}(I, b, I_f)$ is an IPLTS $\langle P', pre', post' \rangle$, where $P' = \langle L', R', B', \sigma' \rangle$, such that:

(i) L' is an LTS $\langle Q', q_0', A', \Delta' \rangle$, where

(a) $Q' = Q \cup Q_f \setminus \{b\}$

(b) $q_0' = \begin{cases} q_0 & \text{if } q_0 \neq b; \\ q_{0,f} & \text{if } q_0 = b; \end{cases}$

(c) $A' = A \cup A_f$;

(d) $\Delta' = \Delta \cup \Delta_f \setminus \Delta(b) \cup \{(\delta^-, \delta_e, q_{0,f}) \mid \delta \in \Delta^-(b)\} \cup \{(q, \delta_e, \delta^+) \mid \delta \in \Delta^+(b) \text{ and } q \in Q_{f,f}\}$;

(ii) $R' = R \cup R_f$;

(iii) $B' = B \cup B_f \setminus \{b\}$;

(iv) $\sigma'(b) = \begin{cases} \sigma(b) & \text{if } q \in B \setminus \{b\}; \\ \sigma_f(b) & \text{if } q \in B_f; \end{cases}$

(v) $pre'(b) = \begin{cases} pre(b) & \text{if } q \in B \setminus \{b\}; \\ pre_f(b) & \text{if } q \in B_f; \end{cases}$

(vi) $post'(b) = \begin{cases} post(b) & \text{if } q \in B \setminus \{b\}; \\ post_f(b) & \text{if } q \in B_f \end{cases}$

Intuitively, the integration of an IPLTS$_f$ I_f in an IPLTS I considering the black-box state b is an IPLTS defined over the LTS L'. The LTS L' contains all the states of the IPLTS I and the IPLTS$_f$ I_f with the exception of the black-box state b (Condition (i).a). The initial state of the integration is the initial state of the IPLTS$_f$ I_f if the black-box state was an initial state, and the initial state of the IPLTS I otherwise (Condition (i).b). The alphabet of the obtained IPLTS is the union of the alphabets of the IPLTS and the IPLTS$_f$ (Condition (i).c). Transitions of the IPLTS C' contain all the transitions of the IPLTS I and the IPLTS$_f$ I_f. However, every incoming transition of b is replaced by a transition with the same initial state and with destination being the initial state of IPLTS$_f$ I_f, and every outgoing transition of b is replaced by a transition with the same final state and with source being the final state of IPLTS$_f$ I_f (Condition (i).d). The set of the regular states of the integrated IPLTS$_f$ C' contains all the regular states of the IPLTS$_f$ I_f and the IPLTS I (Condition (ii)). The set of the black-box states of the integrated IPLTS$_f$ C' contains all the black-box states of the IPLTS$_f$ I_f and the IPLTS I excluding the black-box state b (Condition (iii)). The interfaces of all the black-box states of the IPLTS$_f$ I_f (excluding the black-box state b) and of the IPLTS I, as well as their pre- and post-conditions, are preserved (Conditions (iv), (v) and (vi)).

For example, integrating the sub-component $\mathcal{R}$ for black-box state 2 in Fig. 3e into the partial component $\mathcal{C}$ in Fig. 3d produces the IPLTS shown in Fig. 3j. The prefix "2." is used to identify the states obtained from $\mathcal{R}$.

The contracts of black-box states 4 and 5 are the same as those in Fig. 3c. The initial state of the integration is the initial state of the $\mathcal{C}$ in Fig. 3d. Every incoming transition of state 2 is replaced by a transition with the same initial state and with destination being the initial state 1 of $\mathcal{R}$, and every outgoing transition of 2 is replaced by a transition with the same final state and with source being the final state 5 of the sub-component $\mathcal{R}$.

The integration operator allows replacing a black-box state b of the partial component $\mathcal{C}$ by a sub-component $\mathcal{R}$ and obtaining another partial component $\mathcal{C}'$.

Definition 6.12 Let b be a black-box state, $\mathit{pre}(b)$ and $\mathit{post}(b)$ be its pre-conditions and post-conditions. Let $I = \langle P, \mathit{pre}, \mathit{post}\rangle$ be an IPLTS$_f$, where $P = \langle F, R, B, \sigma\rangle$ and $F = \langle Q, q_0, A, \Delta, Q_f\rangle$ and D be an LTS. The IPLTS$_f$ I_f is *substitutable* for D if for every sequence $\pi_i;\ \pi_e$ where

1. $\pi_i = e_0, e_1, \ldots, e_n$ where $q_0, e_0, q_1, e_1, \ldots, q_n, e_n, q_{n+1}$ is a sequence of the LTS L such that for all i, where $0 \leq i < n$, (q_i, e_i, q_{i+1}) is a transition of the LTS D and $\pi_i \models \mathit{pre}(b)$;
2. π_e is a finite trace of $P \parallel D$, where q_{n+1} is considered an initial state of the LTS;

we have that $\pi_e \models \mathit{post}(b)$.

Intuitively, a sub-component $\mathcal{R}$ is substitutable if, assuming that the environment evolves in a way that respects the pre-condition $\mathit{pre}(b)$, $\mathcal{R}$ ensures the satisfaction of the post-condition.

Correctness. To ensure the correctness of our framework, we show that replacing a substitutable IPLTS$_f$ by a well-formed IPLTS preserves satisfaction of its properties.

Theorem 6.1 Let an IPLTS I with a black-box state b, an IPLTS$_f$ I_f, a property ϕ and an LTS L such that

1. I is well-formed w.r.t. L,
2. I_f is substitutable w.r.t. L, and
3. $I \parallel L \models \phi$

be given. Then, the integration $\textsc{Integrate}(I, b, I_f)$ is such that $\textsc{Integrate}(I, b, I_f) \parallel L \models \phi$.

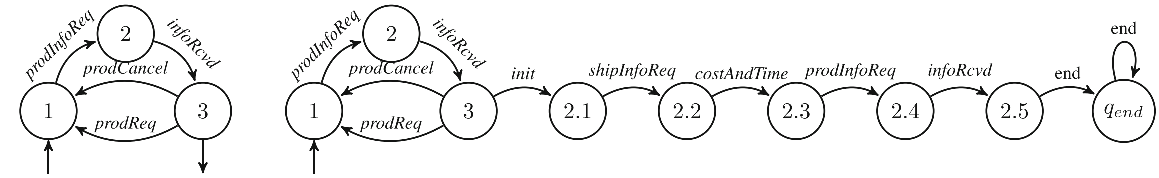
Proof Sketch Since I is well-formed, every valid trace of $S = I \parallel L$ satisfies all the pre-conditions of every black-box state b of I (by Definition 6.8). Since $I \parallel L \models \phi$, if the post-conditions are satisfied, all the infinite traces of $I \parallel L$ satisfy the property ϕ (by Definition 6.9). Since I_f is substitutable, if the pre-precondition of b is satisfied, I_f ensures the satisfaction of the post-condition (Definition 6.12). Thus, by replacing the black-box state b with the IPLTS$_f$ I_f, the satisfaction of the post-conditions is guaranteed, and thus $\textsc{Integrate}(I, b, I_f) \parallel L \models \phi$. □

For example, the sub-component R from Fig. 3e is substitutable; thus, integrating it into the partial component $\mathcal{C}$ shown in Fig. 3j ensures that the resulting integrated component $\mathcal{C}'$ preserves properties *P1-P4*, as intended.

Definition 6.13 Let $F = \langle Q, q_0, A, \Delta, Q_f\rangle$ be an LTS$_f$ and $I_f = \langle D, R, B, \sigma\rangle$ be an IPLTS$_f$ with $D = \langle Q_I, q_{0,I}, A_I, \Delta_I, Q_{f,I}\rangle$. The *sequential composition* of F and I_f, denoted by $F.I_f$, is an IPLTS $\langle L, R \cup Q, B, \sigma\rangle$ where L is the LTS $L = \langle Q \cup Q_I \cup \{q_{end}\}, q_0, A \cup A_I \cup \{\mathit{init}, \mathit{end}\}, \Delta'\rangle$, where $\Delta' = \Delta \cup \Delta_I \cup \{(q, \mathit{init}, q_{0,I}) \mid q \in Q_f)\} \cup \{(q, \mathit{end}, q_{end}) \mid q \in Q_{f,I}\} \cup \{(q_{end}, \mathit{end}, q_{end})\}\}$.

Intuitively, the *sequential composition* of F and I_f is an IPLTS. All the final states of the F are connected to the initial state of the IPLTS$_f$ I_f by a transition labeled with a fresh event *init*. An additional state q_{end} is added to the LTS with a self-loop labeled with *end*. A transition from q to q_{end} labeled with *end* is added for each final state q of the IPLTS$_f$ I_f. Performing these steps ensures that all the infinite traces π of I' are in the form $\pi_1;\mathit{init};\pi_2;\mathit{end}^\omega$, where π_1 is generated by the LTS$_f$ F and π_2 is generated by the IPLTS$_f$ I_f.

Consider for example the LTS$_f$ presented in Fig. 9a which is obtained from the LTS in Fig. 1a by considering the state 3 as a final state and the IPLTS$_f$ of Fig. 3e. The sequential composition is represented in Fig. 9b. Its initial state is the initial state of the LTS$_f$ represented in Fig. 9a. The final state of the LTS$_f$ is connected to the initial state of the IPLTS$_f$ of Fig. 3e with an *init* labeled transition. The final state of the IPLTS$_f$ is connected to the state q_{end} through a transition labeled with *end*. A self-loop labeled with *end* is added to state q_{end}.



(a) LTS$_f$ model of the furniture-sale.

(b) Sequential composition among the LTS$_f$ in Fig. 9a and the IPLTS$_f$ in Fig. 3e.

Fig. 9. Sequential composition: an example

7. Verification algorithms

In this section, we describe the algorithms for the analysis of partial components introduced in Sect. 3, which we have implemented on top of LTSA [MK99]. Section 7.1 describes a method for checking realizability (Algorithm 3). Section 7.2 describes a well-formedness check (Algorithm 4). Section 7.3 describes a model-checking algorithm (Algorithm 5). Finally, Sect. 7.4 describes a method for checking substitutability (Algorithm 6).

7.1. Checking realizability

Let $\mathcal{C}$ be a partial component, $\mathcal{E}$ be its environment and ϕ be a property of interest. The realizability checking algorithm (Algorithm 3) verifies whether there exists an integration (Definition 6.1) of $\mathcal{C}$ obtained by integrating sub-components (Definition 6.11) that constrain the environment $\mathcal{E}$ in a way that satisfies ϕ, i.e., there exists an integration $\mathcal{C}'$ of $\mathcal{C}$ such that $\mathcal{C}' \parallel \mathcal{E} \models \phi$ and there is at least an infinite trace in $\mathcal{C}' \parallel \mathcal{E}$.

Algorithm 3 Checking realizability

```
1: function CHECKREALIZABILITY($\mathcal{E}$, $\mathcal{C}$, $\phi$)
2:     $\mathcal{C}^B \Leftarrow$ REMOVEBOXES($\mathcal{C}$)
3:     if CHECKFLTLONLTS($\mathcal{C}^B \parallel \mathcal{E}$, $\phi$)= false then
4:         return false
5:     if CHECKFLTLONLTS($\mathcal{C} \parallel \mathcal{E}$, $\neg\phi$)=true then
6:         return false
7:     else
8:         return true
```

The algorithm removes all the black-box states and their incoming and outgoing transitions from $\mathcal{C}$ and stores the obtained LTS in $\mathcal{C}^B$ (function REMOVEBOXES on Line 2). It checks whether $\mathcal{C}^B \parallel \mathcal{E} \models \phi$ (Lines 3-4) using the classical CHECKFLTLONLTS model-checking procedure (Algorithm 1). If the property $\neg\phi$ is satisfied, the component is not realizable. Otherwise, it computes $\mathcal{C} \parallel \mathcal{E}$ (as specified in Definition 6.2) and model-checks it against $\neg\phi$ using the classical model-checking procedure CHECKFLTLONLTS, returning the negation of the computed result (Lines 5-8). If CHECKFLTLONLTS($\mathcal{C} \parallel \mathcal{E}$, $\neg\phi$) returns *true*, all the infinite behaviors satisfy the negation of the property ϕ, so there is no infinite trace in $\mathcal{C} \parallel \mathcal{E}$ satisfying ϕ. Since we aim to construct an LTS obtained from $\mathcal{C}$ by integrating sub-components into its black-box states that contain infinite traces that satisfy ϕ, and no infinite traces that satisfy ϕ exist in $\mathcal{C} \parallel \mathcal{E}$, a value *false* is returned. Recall that $\mathcal{C} \parallel \mathcal{E}$ allows the environment $\mathcal{E}$ to perform any transition while $\mathcal{C}$ is in one of its black-box states. If CHECKFLTLONLTS($\mathcal{C} \parallel \mathcal{E}$, $\neg\phi$) returns *false*, there exists an infinite behavior that satisfies ϕ. The integration of the partial component $\mathcal{C}$ that ensures the satisfaction of the property ϕ can be obtained through an integration that forces the environment to only show this behavior.

For example, let $\mathcal{C}$ be the p&d supervisor design (Fig. 3d), $\mathcal{E}$ be the p&d environment (Figs. 1a, 1b and 1c) and the property *P2* (Fig. 1d). The realizability checker confirms the existence of an integration that satisfies property *P2*.

Since the algorithm simply calls the CHECKFLTLONLTS twice, its complexity is exponential in the size of the FLTL formula and linear in the size of the partial component $\mathcal{C}$.

Theorem 7.1 Let $\mathcal{C}$ be a partial component, $\mathcal{E}$ be its environment and ϕ be a property of interest. Algorithm 3 returns *false* if there is no component $\mathcal{C}'$ obtained from $\mathcal{C}$ by integrating sub-components, s.t. $\mathcal{C}' \parallel \mathcal{E} \models \phi$.

Proof Sketch If $\mathcal{C}^B \parallel \mathcal{E} \not\models \phi$, there exists an infinite trace of the environment which is allowed by the already specified portion of the component and which violates ϕ. No matter how the sub-components of the black-box states are specified, there will be an infinite trace that does not satisfy ϕ. Thus, the component is not realizable.

If $\mathcal{C} \parallel \mathcal{E} \models \neg\phi$, there does not exist an infinite trace on which ϕ holds. Thus, there cannot exist a component that constrains the environment to expose only traces that satisfy ϕ and ensures that at least one infinite trace is present in the final system. □

7.2. Checking well-formedness

Let $\mathcal{C}$ be a partial component and $\mathcal{E}$ be its environment. The well-formedness checker (Algorithm 4) verifies whether the partial component $\mathcal{C}$ is well formed w.r.t. its environment $\mathcal{E}$ (Definition 6.8).

Algorithm 4 Checking well-formedness

```
1:  function CHECKWELL- FORMED($\mathcal{E}$, $\mathcal{C}$)
2:      for $b \in B$ do
3:          $\mathcal{C}' \Leftarrow \mathcal{C}$
4:          for $q \in B \setminus b$ do
5:              $\mathrm{LTS}_f(q) \Leftarrow \mathrm{FLTL}_f\mathrm{2LTS}_f(\mathrm{post}(q))$
6:              $\mathcal{C}' \Leftarrow$ INTEGRATE($\mathcal{C}'$, $q$, $\mathrm{LTS}_f(q)$)
7:          $\mathrm{LTS}_f(b) \Leftarrow \mathrm{FLTL}_f\mathrm{2LTS}_f(\mathrm{post}(b))$
8:          $\langle \mathcal{C}'', q_{0,b} \rangle$ = INTEGRATE$^*$($\mathcal{C}'$, $b$, $\mathrm{LTS}_f(b)$)
9:          $S = \mathrm{LTS2LTS}_f(\mathcal{C}'', q_{0,b}, \mathcal{E})$
10:         if (CHECKFLTL$_f$ONLTS$_f$($S$, $pre(b)$)=$false$) then
11:             return $false$
        return $true$
```

Algorithm 4 considers each black-box state b of the partial component $\mathcal{C}$ (Line 2). For each b, it creates a copy $\mathcal{C}'$ of the partial-component (Line 3). For each block-box state q, different from b, it transforms its post-condition $post(q)$ into an LTS_f denoted $\mathrm{LTS}_f(q)$ using the procedure $\mathrm{FLTL}_f\mathrm{2LTS}_f$ described in Algorithm 2 (Line 5). This LTS_f is integrated into $\mathcal{C}'$ (Line 6). Then, the algorithm transforms the post-condition $post(b)$ of b into an LTS_f denoted $\mathrm{LTS}_f(b)$ (Line 7) and integrates it into $\mathcal{C}'$ (Line 8). The operator INTEGRATE* behaves like the usual INTEGRATE operator (Definition 6.11) but in addition returns the initial state $q_{0,b}$ of the post-condition $post(b)$. Then, we cast the $\mathcal{C}''$ over $\mathcal{E}$ w.r.t. the initial state $q_{0,b}$ of the post-condition (Definition 6.5). Finally, the algorithm checks the satisfaction of the precondition $pre(b)$ of b on the LTS_f S by using the procedure CHECKFLTL$_f$ONLTS$_f$ described in Sect. 5.2 (Line 10).

In the p&d example, if we remove the clause $\Diamond$ *F_InfoRcvd* from the post-condition of the black-box state 2, the p&d component is not well-formed since the pre-condition of state 4 is violated. The counterexample shows a trace that reaches the black-box state 4 in which an event *userReq* is not followed by *infoRcvd*. Adding $\Diamond$ *F_InfoRcvd* to the post-condition of state 2 solves the problem.

Algorithm 4 calls the method CHECKFLTL$_f$ONLTS$_f$ with the pre-condition $pre(b)$ and an LTS obtained from $\mathcal{C}$ by integrating a set of LTS_f, one for each black-box state, generated from their post-conditions. In the worst case, the size of each LTS_f is exponential in the size of the corresponding post-condition. Thus, the complexity is exponential in the size of the FLTL_f formula of the post-conditions and in the size of the FLTL_f pre-condition, and linear in the size of the partial component $\mathcal{C}$.

Theorem 7.2 Let $\mathcal{C}$ be a partial component and $\mathcal{E}$ be its environment. The well-formedness procedure returns *true* if and only if $\mathcal{C}$ is well-formed (Definition 6.8).

Proof Sketch By construction, the finite traces of $\mathrm{LTS2LTS}_f(\mathcal{C}'', q_{0,b}, \mathcal{E})$ are exactly the finite traces that satisfy the post-conditions of the black-box states (Definition 6.8). Thus, the procedure CHECKFLTL$_f$ONLTS$_f$ returns *false* if and only if there exists a finite trace π of $\mathcal{C} \parallel \mathcal{E}$ that does not satisfy the pre-condition $pre(b)$ (Definition 6.7). □

7.3. Model checking

Let $\mathcal{C}$ be a well-formed partial component, $\mathcal{E}$ be its environment and ϕ be a property of interest. Model checking (Algorithm 5) verifies whether $\mathcal{C}$ satisfies the property of ϕ under the environment $\mathcal{E}$ (Definition 6.9).

Algorithm 5 Model checking

```
1: function CHECKMODEL($\mathcal{E}$, $\mathcal{C}$ $\phi$)
2:     $\mathcal{C}' \Leftarrow \mathcal{C}$
3:     for $q \in B$ do
4:         $\text{LTS}_f(q) \Leftarrow \text{FLTL}_f\text{2LTS}_f(post(q))$
5:         $\mathcal{C}' \Leftarrow \text{INTEGRATE}(\mathcal{C}', q, \text{LTS}_f(q))$
6:     return CHECKFLTLONLTS($\mathcal{C}' \parallel \mathcal{E}, \phi$)
```

Algorithm 5 first creates a copy $\mathcal{C}'$ of the partial component $\mathcal{C}$ (Line 2), which is then iteratively modified by the algorithm (Line 5). Specifically, the algorithm transforms all post-conditions into an LTS_f (Line 4) and integrates the generated LTS_f into the copy $\mathcal{C}'$ of the partial component (Line 5). Since all of the black-box states are replaced by the corresponding LTS_f, the obtained model of the partial component $\mathcal{C}'$ is an actual LTS. Finally, the algorithm uses the CHECKFLTLONLTS model-checking procedure (Algorithm 1) to verify whether $\mathcal{C}' \parallel \mathcal{E} \models \phi$ (Line 6).

If we consider the design in Fig. 3d and assume that the black-box state 2 is not associated with any post-condition, then the model checker returns a counterexample *userReq*, τ, *offerRcvd* for property *P2*, since the sub-component that will replace the black-box state 2 is not forced to ask to book the furniture service. Adding the post-condition in Fig. 3c solves the problem.

Algorithm 5 calls the CHECKFLTLONLTS method considering the property ϕ and an LTS that is obtained from $\mathcal{C}$ by integrating a set of LTS_f, one for each black-box state, that are generated from their post-conditions. The size of each of these LTS_f is in the worst case exponential in the size of the corresponding post-condition. Thus, the complexity is exponential in the size of the FLTL property ϕ, and in the size of the post-conditions. It is linear in the size of the partial component $\mathcal{C}$.

Theorem 7.3 Let $\mathcal{E}$ be an environment and let $\mathcal{C}$ be a partial component well-formed over $\mathcal{E}$, and ϕ be a property of interest. The model-checking procedure returns *true* if and only if $\mathcal{C}$ satisfies the property ϕ (Definition 6.9).

Proof Sketch The model-checking procedure runs the CHECKFLTLONLTS algorithm by considering the LTS $\mathcal{C}' \parallel \mathcal{E}$. $\mathcal{C}'$ is obtained from the partial component $\mathcal{C}$ by integrating the LTS_f generated from the post-conditions of its black-box states. Thus, $\mathcal{C}' \parallel \mathcal{E}$ contains exactly the traces of $\mathcal{C} \parallel \mathcal{E}$ that satisfy the post-conditions of the black-box states of $\mathcal{C}$ (Definition 6.7). Furthermore, since $\mathcal{C}$ is well-formed, by Definition 6.9, ϕ holds. □

7.4. Checking substitutability

Let $\mathcal{E}$ be an environment, $\mathcal{R}$ be a sub-component for a black-box state b with interface $\sigma(b)$, pre-condition *pre*(b) and post-condition *post*(b). Algorithm 6 checks whether the sub-component $\mathcal{R}$ is substitutable (Definition 6.12).

Algorithm 6 Substitutability checking

```
function CHECKSUBST($\mathcal{E}$, $\mathcal{R}$, $b$, $\sigma(b)$, $pre(b)$, $post(b)$)
    for $q \in B$ do
        $\text{LTS}_f(q) \Leftarrow \text{FLTL}_f\text{2LTS}_f(post(q))$
        $\mathcal{C}' \Leftarrow \text{INTEGRATE}(\mathcal{C}', q, \text{LTS}_f(q))$
    $\text{LTS}_f(b) = \text{FLTL}_f\text{2LTS}_f(pre(b))$
    $P = \text{LTS}_f(b).\mathcal{R}$
    $\lambda = \Box(\text{INIT} \rightarrow \bigcirc(post(b)))$
    return CHECKFLTLONLTS(($P \parallel \mathcal{E}$), $\lambda$)
```

Algorithm 6 transforms all the post-conditions in LTS_f using the procedure $\text{FLTL}_f\text{2LTS}_f$ and integrates them (Lines 2-4). Then, it transforms the pre-condition $pre(b)$ into an LTS_f (Line 5) and computes the sequential composition between the computed LTS_f and the sub-component $\mathcal{R}$ (Line 6). The post-condition $post(b)$ is transformed into the FLTL formula $\lambda = \Box(\text{INIT} \rightarrow \bigcirc(\text{post}(b)))$, i.e., the post-condition must hold after the fluent INIT becomes *true*. Finally, it verifies the satisfaction of the transformed post-condition on the sequential composition (Line 8) by executing the function CHECKFLTLONLTS on the parallel composition $P \parallel \mathcal{E}$ and the modified post-condition λ.

In the p&d example, the substitutability checker does not return any counterexample for the sub-component in Fig. 3e. Thus, the sub-component can be integrated in place of the black-box state 2.

Algorithm 6 calls the CHECKFLTLONLTS method considering the property λ and an LTS $P \parallel \mathcal{E}$. The IPLTS P is obtained from the sub-component $\mathcal{R}$ by integrating a set of LTS_f, one for each black-box state, that are generated from their post-conditions, and by computing the sequential composition with an LTS_f generated from the pre-condition $pre(b)$. The size of each of the integrated LTS_f is in the worst case exponential in the size of the corresponding post-condition. The size of the LTS_f generated from the pre-condition $pre(b)$ is in the worst case exponential in the size of the pre-condition. Thus, the complexity is exponential in the size of the FLTL_f property ϕ, of the post-conditions of the black-box states contained in the sub-component and of its pre-condition $pre(b)$. It is linear in the size of the partial component $\mathcal{C}$.

Theorem 7.4 Let $\mathcal{E}$ be an environment, $\mathcal{R}$ be a sub-component for a black-box state b with interface $\sigma(b)$, pre-condition $pre(b)$ and post-condition $post(b)$. The CHECKSUBST procedure returns *true* if and only if the sub-component $\mathcal{R}$ is substitutable (Definition 6.12).

Proof Sketch By construction, the LTS_f $P \parallel \mathcal{E}$ contains traces of the form $\pi_i;\ \pi_e$ that satisfy the conditions of Definition 6.12. By checking whether λ holds on $P \parallel \mathcal{E}$ using the procedure $\text{CHECKFLTL}_f\text{ONLTS}_f$, we verify that $\pi_e \models post(b)$. □

8. Tool support and evaluation

This section reports on our experience evaluating the effectiveness and scalability of our approach. Specifically, we aim to answer the following questions:

RQ1: How *effective* is FIDDle w.r.t. supporting an iterative, distributed development of correct controllers? (Sect. 8.2) and

RQ2: How *scalable* is the automated part of the proposed approach? (Sect. 8.3).

We begin by describing our tool support.

8.1. Tool support

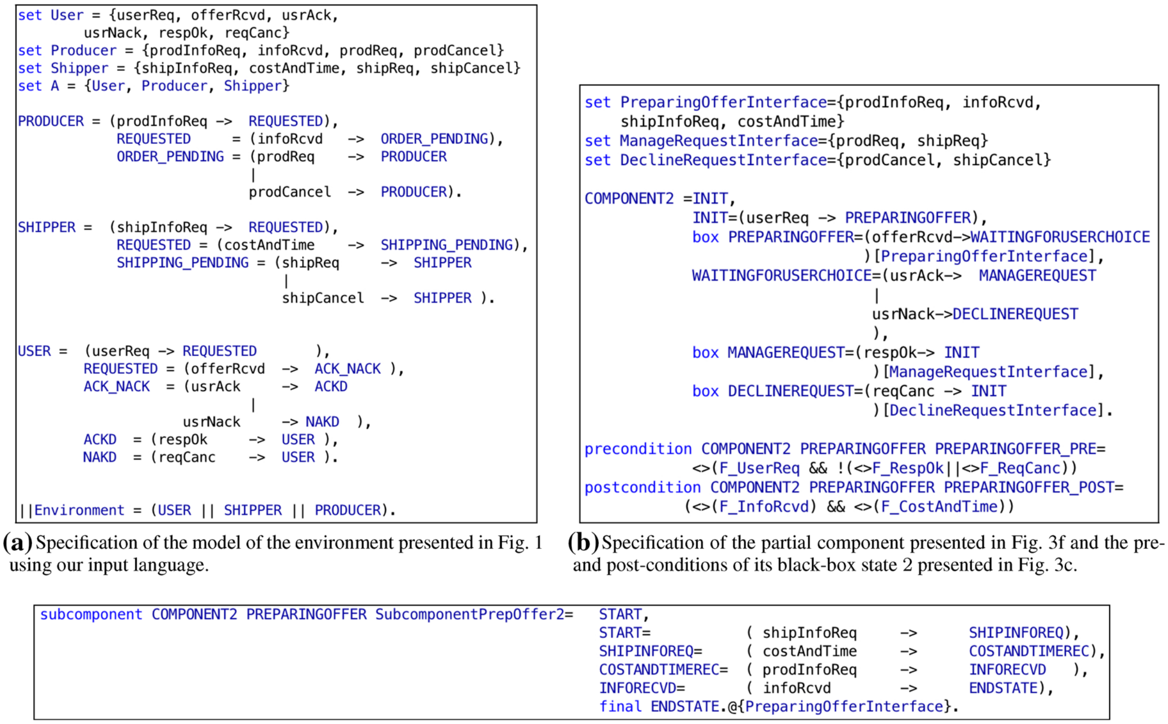
FIDDle is a Java application developed on top of LTSA to provide support for incremental distributed development of controllers. The complete implementation together with the examples and the case study presented in the next section is available at https://github.com/claudiomenghi/FIDDLE. FIDDle extends LTSA in three directions:

- input language and compiler: extended to support modeling of partial components and sub-components;
- graphical interface: extended to provide graphical support to users;
- verification algorithms: implemented the procedures described in Sect. 7.

We describe each of these directions below.

8.1.1. Input language

We introduced a set of constructs within the LTSA input language that support the formalism defined in Sect. 6. These commands (see Table 1) introduce new keywords to define partial components (**`partial component`**), their black-box states (**`box`**), their pre- and post-conditions (**`precondition`**, **`postcondition`**), the environment (**`environment`**), and the sub-components (**`subcomponent`**).



```
set User = {userReq, offerRcvd, usrAck,
        usrNack, respOk, reqCanc}
set Producer = {prodInfoReq, infoRcvd, prodReq, prodCancel}
set Shipper = {shipInfoReq, costAndTime, shipReq, shipCancel}
set A = {User, Producer, Shipper}

PRODUCER = (prodInfoReq ->  REQUESTED),
            REQUESTED     = (infoRcvd   ->  ORDER_PENDING),
            ORDER_PENDING = (prodReq    ->  PRODUCER
                            |
                            prodCancel  ->  PRODUCER).

SHIPPER =  (shipInfoReq ->  REQUESTED),
            REQUESTED = (costAndTime    ->  SHIPPING_PENDING),
            SHIPPING_PENDING = (shipReq     ->  SHIPPER
                                |
                                shipCancel  ->  SHIPPER ).


USER =  (userReq -> REQUESTED       ),
        REQUESTED = (offerRcvd  ->  ACK_NACK ),
        ACK_NACK  = (usrAck     ->  ACKD
                               |
                    usrNack     -> NAKD  ),
        ACKD  = (respOk     ->  USER ),
        NAKD  = (reqCanc    ->  USER ).


||Environment = (USER || SHIPPER || PRODUCER).
```

(a) Specification of the model of the environment presented in Fig. 1 using our input language.

```
set PreparingOfferInterface={prodInfoReq, infoRcvd,
    shipInfoReq, costAndTime}
set ManageRequestInterface={prodReq, shipReq}
set DeclineRequestInterface={prodCancel, shipCancel}

COMPONENT2 =INIT,
            INIT=(userReq -> PREPARINGOFFER),
            box PREPARINGOFFER=(offerRcvd->WAITINGFORUSERCHOICE
                                )[PreparingOfferInterface],
            WAITINGFORUSERCHOICE=(usrAck->  MANAGEREQUEST
                                 |
                                 usrNack->DECLINEREQUEST
                                 ),
            box MANAGEREQUEST=(respOk-> INIT
                                 )[ManageRequestInterface],
            box DECLINEREQUEST=(reqCanc -> INIT
                                 )[DeclineRequestInterface].

precondition COMPONENT2 PREPARINGOFFER PREPARINGOFFER_PRE=
            <>(F_UserReq && !(<>F_RespOk||<>F_ReqCanc))
postcondition COMPONENT2 PREPARINGOFFER PREPARINGOFFER_POST=
           (<>(F_InfoRcvd) && <>(F_CostAndTime))
```

(b) Specification of the partial component presented in Fig. 3f and the pre- and post-conditions of its black-box state 2 presented in Fig. 3c.

```
subcomponent COMPONENT2 PREPARINGOFFER SubcomponentPrepOffer2=   START,
                                                                 START=          ( shipInfoReq    ->      SHIPINFOREQ),
                                                                 SHIPINFOREQ=    ( costAndTime    ->      COSTANDTIMEREC),
                                                                 COSTANDTIMEREC= ( prodInfoReq    ->      INFORECVD    ),
                                                                 INFORECVD=      ( infoRcvd       ->      ENDSTATE),
                                                                 final ENDSTATE.@{PreparingOfferInterface}.
```

(c) Specification of the sub-component presented in Fig. 3e.

Fig. 10. Specification of the p&d motivating example using our extended syntax

For example, the model of the environment of our motivating example presented in Fig. 1, the partial component presented in Fig. 3f and the pre- and post-conditions of the black-box state 2 presented in Fig. 3c and the sub-component presented in Fig. 3e are described using the proposed keywords given in Fig. 10.

8.1.2. Graphical user interface

We modified the graphical interface of LTSA (Fig. 11) in three different directions:

1. the commands described in Table 1 are integrated within the textual interface embedded within LTSA, including syntax highlighting;
2. the LTSA GUI is extended to visualize partial components and sub-components, with black-box states identified by black-colored states, as shown in Fig. 11 (①);
3. a new menu is added to the interface to enable loading of partial components, sub-components and their environments (② in Fig. 11);
4. a new menu is added to allow running the checks described in Sect. 7 (③ in Fig. 11).

8.1.3. Verification algorithms

The verification algorithms presented in Sect. 7, as well as Algorithm 2 and the function FLTL$_f$2FLTL presented in Sect. 5 have been developed within appropriate Java classes. The implementation reused existing LTSA classes, as specified in the description of the appropriate algorithm. However, implementation was not straightforward. Reverse engineering the existing LTSA classes, which were written by different authors and that were often not completely documented, took a considerable effort. Thus, integrating the existing algorithms within the existing classes was not easy and required more than 1000 lines of Java code.

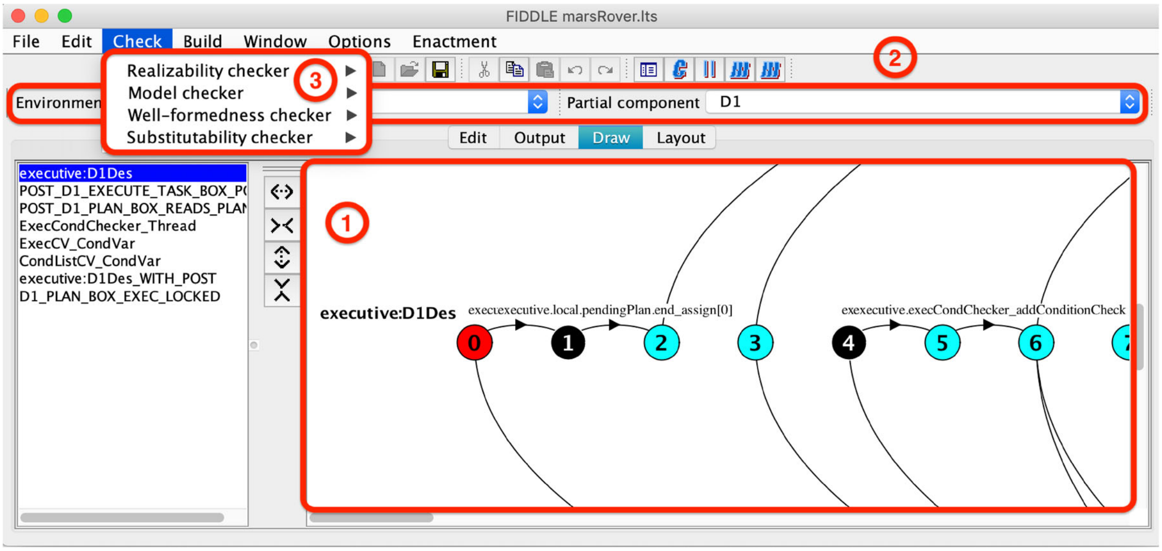


Fig. 11. FIDDle graphical user interface

Table 1. Syntax of the new commands added in LTSA to support FIDDle

Syntax	Description
box name = (trans)[interface]	name: the name of the box, trans: the outgoing transitions of the box, interface: the set of events in the interface of the black-box state.
partialcomponent name = proc	name: the name of the partial component, proc: the description of the process of the partial component.
subcomponent comp box name=proc **final** **ENDSTATE.@**{interface}.	defines a subcomponent: comp: the partial component for which the sub-component is designed, box: the name of the box of the partial component for which the sub-component is designed, name: the name of the sub-component, proc: the definition of the sub-component behavior, ENDSTATE: indicates the terminal state of the sub-component, interface: contains the interface of the sub-component.
precondition comp box name=formula	comp: the partial component that contains the black-box state, box: the box of the partial component to which the pre-condition is associated, name: the name of the precondition, formula: the FLTL$_f$ formula of the precondition using the standard LTSA FLTL syntax.
postcondition comp box name=formula	comp: the partial component that contains the black-box state, box: the box of the partial component to which the post-condition is associated, name: the name of the post-condition, formula: the FLTL$_f$ formula of the post-condition using the standard LTSA FLTL syntax.

8.2. Effectiveness

In this section, we describe a case study which aims to show how the proposed approach supports developers in the initial overall design of the controller and in iterative and distributed development of (sub-)components. To do so, we simulated forward development of an existing complex controller and analyzed FIDDle-provided support along the steps described in Sect. 3.

8.2.1. *Experimental setup*

We chose the executive module of the K9 Mars Rover, developed at NASA Ames [GPB02, CGP03, GPB05] and specified using LTS. The overall size of the LTS is $\tilde{1}0^7$ states.

Table 2. High-level description of the Mars Rover components

Component	Description	#States
ExecCondChecker	Designed to monitor the state conditions.	3300
ActionExecution	Responsible for issuing the commands to the Rover. It receives an action as input and checks whether it is already under execution. In this case, it stops the action by sending an interrupt and then executes the action. If no action is under execution, it executes it.	45
DBMonitor	Allows reading and writing informations.	13
Executive (D3)	The main coordinating component. The complete model of the Executive component comprised of 11 states, *WaitingForPlan*, *PrepareExecution*, *ExecutePlan*, *ExecuteCurrentNode*, *ExecuteTaskAction*, *AddTerminatingCondition*, *WaitForTermination*, *CheckSuccessfulCommand*, *CheckExecSignal*, *ClearConditions* and *Done*, each further decomposed as an LTS. The final model of the Executive component was obtained by replacing these states with the corresponding LTS. This model had about 100 states.	11
D2	Obtained from the Executive component by encapsulating states *WaitingForPlan*, *PrepareExecution* into the black-box state *PlanBox*.	9
D1	Obtained from D2 by setting *ExecuteTaskAction* as a black-box state.	9

Table 3. High-level description of the Mars Rover Shared Variables

Mutex	Description
exec	Locked and unlocked by the ExecCondChecker module.
conditionList	Locked and unlocked by the ExecCondChecker module.
action	Locked and unlocked by the ActionExecution module.
db	Locked and unlocked by the DBMonitor module.

The executive module was decomposed into the following components: Executive, ExecCondChecker, ActionExecution and Database. ExecCondChecker was further decomposed into DBMonitor and Internal. A high-level description of each component is provided in Table 2.

Each of these components was associated with a shared variable (*exec*, *conditionList*, *action* and *db*) used to communicate with the other components, e.g., the *exec* variable was used by ExecCondChecker to communicate with Executive. Access to each shared variable was done via mutexes, documented in Table 3.

We considered two properties:

Pr1: Executive performed an action only after a new plan was read from Database;

Pr2: Executive got the lock over the *condList* variable only after obtaining the *exec* lock.

The FLTL formulae for the properties of interest are presented in Table 4. To specify property Pr1, we defined two fluents: EXEC_ACTION, which is *true* while the action is under execution, and READ_PLAN, which is *true* while the plan is read. Fluents are detailed in Table 5. The formalization of property Pr1 specifies that an action is not performed before a plan is read.

To specify property Pr2, we defined fluents, COND_LIST_LOCKED and EXEC_LOCKED. These fluents are *true* after the *condList* and the *exec* are locked and until it is unlocked. The formalization of this property specifies that if the condition list is locked, the *exec* is also locked.

We simulated forward development of the Executive component. We considered the existing model (D3) of the Executive and abstracted portions of the complete model into black-box states to create two partial components, D1 and D2, representing partial designs. The abstracted portions are used to generate sub-components Sub1 and Sub2 for the black-box states contained in the partial design which we assumed third-party companies had to develop.

Table 4. Formalization of the considered properties and pre- and post-conditions in FLTL and FLTL$_f$

Element	Formula
Pr1	$(\neg \text{EXEC\_ACTION})\,\mathcal{W}(\text{READ\_PLAN} \wedge \neg \text{EXEC\_ACTION})$
Pr2	$\Box(\text{COND\_LIST\_LOCKED} \rightarrow \text{EXEC\_LOCKED})$
Post1	$\Diamond(\text{READ\_PLAN})$
Post2	$\Box(\text{COND\_LIST\_LOCKED} \rightarrow \text{EXEC\_LOCKED})) \wedge (\Diamond(\Box \text{EXEC\_LOCKED})))$
Post3	$\Box(\text{EXEC\_ACTION} \rightarrow \text{EXEC\_LOCKED}) \wedge \Diamond\Box(\text{EXEC\_LOCKED} \wedge \neg \text{EXEC\_ACTION})$
Pre1	$\Diamond\Box(\neg \text{EXEC\_ACTION} \wedge \text{EXEC\_LOCKED})$
Pre2	$(\Diamond\Box(\text{EXEC\_LOCKED} \wedge \neg \text{COND\_LIST\_LOCKED})) \wedge (\Box(\text{COND\_LIST\_LOCKED} \rightarrow \text{EXEC\_LOCKED})))$

Table 5. Fluents and their description

Fluent name	Formal definition	Textual description
READ_PLAN	$\langle\{begin\_read\}, \{end\_read\}, false\rangle$	*true* from the moment in which EXECCONDCHECKER starts reading the plan to the moment in which the plan is read.
EXEC_ACTION	$\langle\{install\}, \{action.unlock\}, false\rangle$	*true* from the moment in which a new action is installed by EXECUTIVE, to the moment in which the action lock is released.
COND_LIST_LOCKED	$\langle\{condList.lock\}, \{condList.unlock\}, false\rangle$	*true* while the condition list is locked.
EXEC_LOCKED	$\langle\{exec.lock\}, \{exec.unlock\}, false\rangle$	*true* while *exec* is locked.

To create D2, we encapsulated three states that receive plans and prepare for plan execution into one black-box state *Read_Plans*. The abstracted portion of the EXECUTIVE leads to the sub-component SUB2. To create D1, we also designated one of the 10 states of the EXECUTIVE whose corresponding LTS is in charge of executing a plan, state *ExecuteTaskAction*, as a black-box state. A portion of the design D1 of EXECUTIVE is presented in Fig. 11. The abstracted portion of EXECUTIVE leads to the sub-component SUB1. D2 can be obtained from D1 by integrating the sub-component SUB1 using the same procedure. D3 is obtained from D1 by integrating the sub-component SUB2.

We considered the (partial) designs D1, D2 and D3 and the sub-components SUB1 and SUB2 and used FIDDle to iteratively develop and check their designs. The obtained results are summarized in Table 6 which contains an incremental identifier used as a reference in the textual description (i.e., ID1, ID2, ..., ID15), the check that has been performed, the considered partial component or sub-component, the considered property and pre- and post- conditions. For each check, we specify whether the check succeeded (✓) or fail (✗), the number of states explored by FIDDle, the memory consumption and the time needed to provide the reported results.

Creating an Initial Component Design. The *realizability checker* (Algorithm 3) confirmed the existence of a component obtained from D1 by integrating sub-components that satisfy the properties of interest (ID1 in Table 6). Specifically, for property PR1, FIDDle returned a trace of D1 $\parallel \mathcal{E}$ that satisfies the property of interest, i.e., a trace in which EXEC_ACTION does not occur before READ_PLAN. For property PR2, FIDDle returned a trace of D1 $\parallel \mathcal{E}$ that satisfies the property of interest, i.e., a trace in which EXECUTIVE got the lock over the *condList* variable only after obtaining the *exec* lock (ID2).

The *model checker* (Algorithm 5) returned a counterexample for both properties of interest. For property PR1, it returned a counterexample in which no plan was read and yet an action was performed (ID3). For property PR2, the counterexample was where EXECUTIVE got the *condList* lock without possessing the *exec* lock (ID4).

To guarantee the satisfaction of PR1, we specified the post-condition POST1 defined in Table 4 for the black box state *Read_Plans* (ID5). The post-condition ensures that a plan is read in the black-box state *PlanBox*, since the fluent EXEC_ACTION can only hold after the black-box state *PlanBox* is left. By ensuring that a plan is read in the black-box state *PlanBox*, the property is satisfied.

We added the post-condition POST2 to the black box state *PlanBox* and the post-condition POST3 to the black box state *ExecuteTaskAction* (see Table 4). The post-condition POST2 specifies that if the *conditionList* variable is locked, the *exec* variable is also locked. Furthermore, it forces the *exec* variable to be locked when the component is exited. The post-condition POST3 specifies that if an action is under execution, the *exec* variable should be locked. Furthermore, it forces the *exec* variable to be locked when the component is exited and the action to not be under execution. The model checker confirmed that the new design guaranteed satisfaction of property PR2 (ID6).

We added a pre-condition that specifies that an action is not under execution when the black-box state *Read_Plans* is entered. The *well-formedness checker* (Algorithm 4) confirmed that the pre-condition is ensured by the current design (ID7). We additionally added a pre-condition for the black-box state *PlanBox* specifying that, when the black-box state is entered, the *exec* variable is locked, while the *conditionList* variable is not and the property is not violated before entering the black-box. The *well-formedness checker* (Algorithm 4) confirmed that the pre-condition is ensured by the current design (ID8).

Developing and Integrating the *ExecuteTaskAction* Sub-component. We simulated a refinement process in which pre- and post-conditions were given to third parties for sub-component development. We considered the sub-component SUB1 containing the portion of the EXECUTIVE component abstracted by the black-box state *ExecuteTaskAction*. We ran the *substitutability checker* (ID9) to verify, affirmatively, whether the sub-component SUB1 ensured the post-condition POST3 given the pre-condition PRE1.

Then, we integrated the sub-component SUB1 into the partial design D1 obtaining the partial component D2. We ran the *realizability checker* (ID10) which confirmed the existence of a model that refines D2 and satisfies the properties of interest. Then, we ran the *model checker* which confirmed satisfaction of the property PR1 (ID11) and PR2 (ID12). This shows that the development of the sub-component SUB1 can be distributed to a third party.

Table 6. Checks performed using FIDDle during the iterative development of the Executive component: **ID**, incremental identifier used as reference in the textual description; **Check**, the check being performed; **Comp.**, the considered partial component or sub-component; **Prop.**, the considered property; **Pre**, the considered pre-condition; **Post**, the considered post-condition; **#States**, the number of states explored by FIDDle; **Mem.(Mb)**, the memory consumption of FIDDle; **Time(ms)**, the time needed to produce the reported result; **Result**, did the check succeed (✓) or failed (✗)?

ID	Check	Comp.	Prop.	Pre	Post	#States	Mem.(Mb)	Time(ms)	Result
ID1	*Realizability*	D1	Pr1	-	-	889924	2002193	191611	✓
ID2	*Realizability*	D1	Pr2	-	-	943954	943954	126054	✓
ID3	*Model*	D1	Pr1	-	-	442343	662590	6797	✗
ID4	*Model*	D1	Pr2	-	-	442343	1197398	2140	✗
ID5	*Model*	D1	Pr1	-	Post1	2920400	1384993	61649	✓
ID6	*Model*	D1	Pr2	-	Post1, Post2, Post3	3214400	913841	77701	✓
ID7	*Well-Formedness*	D1	-	Pre1	Post1, Post2, Post3	3116416	2022787	111869	✓
ID8	*Well-Formedness*	D1	-	Pre1, Pre2	Post1, Post2, Post3	3459400	1047821	158076	✓
ID9	*Substitutability*	Sub1		Pre1	Post3	8467200	1345867	307224	✓
ID10	*Realizability*	D2	-	-	Post1, Post2	9955311	2609186	579786	✓
ID11	*Model*	D2	Pr1	-	Post1, Post2	4537400	1701396	139013	✓
ID12	*Model*	D2	Pr2	-	Post1, Post2	4586400	1309030	132629	✓
ID13	*Substitutability*	Sub2	-	Pre2	Post1, Post2	3633840	1073350	92165	✓
ID14	*Model*	D3	Pr1			4321800	1167794	120314	✓
ID15	*Model*	D3	Pr2			4272800	923968	121174	✓

As formally proven in Sect. 6.4 and confirmed by the model checker, integrating a substitutable sub-component into a well-formed partial-component ensures that the satisfaction of the property of interest is preserved.

Developing and Integrating the PlanBox Sub-component. We simulated the iterative refinement of the sub-component *PlanBox*. Distributing pre- and post-conditions allow parallel development of the *PlanBox* and *ExecuteTaskAction* sub-components. We assumed that the developer team provides the design of the sub-component Sub2 abstracted from the Executive component by the black-box state *PlanBox*. Then, in order to verify that the sub-component Sub2 ensured the post-condition Post2 and Post3, we ran the *substitutability checker* (ID13). The check was affirmative showing that the proposed design can replace the black-box state *PlanBox*.

Then, we integrated the sub-component Sub2 into the partial design D2 obtaining the partial component D3. We ran the *model checker* which confirmed satisfaction of the properties Pr1 (ID14) and Pr2 (ID15). This shows that the development of the sub-component Sub2 can be distributed to a third party. and that integrating a substitutable sub-component into a well-formed partial-component ensures that the satisfaction of the property of interest is preserved, as desired.

8.2.2. *Discussion and threats to validity*

The Executive component is a realistic controller of medium size [AFT08, LMP08, BBKT04]. It has been implemented by 25K lines of C++ code, 10K of which is the main control code, and the rest define data structures needed for communication with the actual Rover [GPC04].

FIDDle was effective in analyzing partial components and helping change their design to ensure the satisfaction of the properties of interest. The experiment confirmed the possibility of distributing the design of sub-components for the black-box states. As expected, no rework at the integration level was required, i.e., integration produced components that satisfied the properties of interest. This confirmed that FIDDle supports verification-driven iterative and distributed development of components.

A threat to construct validity concerns the (manual) construction of the intermediate model produced by us by abstracting an existing component model and the design of the properties to be considered. However, the intermediate partial designs and the selected properties were based on original developer comments present in the model. Specifically, the intermediate partial designs were obtained by encapsulating states of the initial model that abstracted portions of the LTS into black-box states. In terms of the properties, the designer's interest in whether the *condList* is locked when actions are performed by Executive is confirmed by the presence of the following comment:

```
AssumedByXXX = (        condList.lock
```

that preceded an instruction in which the Executive modifies the value of a condition of the ExecCondChecker component.

The property specifying that an action is not performed before a plan is read follows from the description of the Mars Rover Executive behavior [GPB02]:

```
The executive receives flexible plans from a Planner, which it
executes according to the plan language semantics. A plan is a
hierarchical structure of actions that the Rover must perform.
```

A threat to internal validity concerns the design of the contracts (pre- and post- conditions and interfaces) for the black-box states chosen along the process. Writing pre- and post-conditions may be difficult and error-prone for practitioners. However, we envision a process in which developers use property specification patterns [DAC98] to define pre- and post-conditions. Patterns collect recurrent specification problems and can guide developers in writing meaningful contracts. Patterns also contain solutions, expressed in temporal logic, for the recurrent specification problems, which can be used as template formulae for pre- and post-conditions, thus facilitating writing correct FLTL_f specifications. That was the method we used ourselves in writing pre- and post-conditions in the reported case study.

The fact that a single example has been considered is a threat to external validity. However, we believe that the presented results can be considered as representative for real case scenarios since the analyzed example is a medium-sized complex real case study [Lev87, Sof04, BBKT04].

8.3. Scalability

We studied the scalability of the procedures proposed in this work. We considered the *well-formedness* and the *substitutability checkers* since the *realizability* and the *model checker* are implemented by performing a call to a classical model checker after simple transformations of the considered models. We set up experiments Exp1 and Exp2 to evaluate the scalability of the *well-formedness* and the *substitutability checkers*, respectively. Our experiments were based on a set of *randomly-generated* models. Random model generation is a widespread technique to evaluate artifacts in the software engineering and formal methods communities [TV05, DWDHR06, TV07, SFG+12, FSC12a, MSG16, MSCG18, MGPT18a, MGPT18b].

Experiment Exp1. We compared performance of the *well-formedness checker* with the standard model checker for FLTL properties on LTS models implemented in LTSA by varying the size of the partial components and their environments. We generated an LTS model of the environment and a complete model of the component and checked the parallel composition between the component and the environment w.r.t. a property of interest using LTSA. Then we generated a partial component (by marking one of the states of the complete component as a black-box and defining pre- and post- conditions for it) and ran the well-formedness checker, comparing performance of the two.

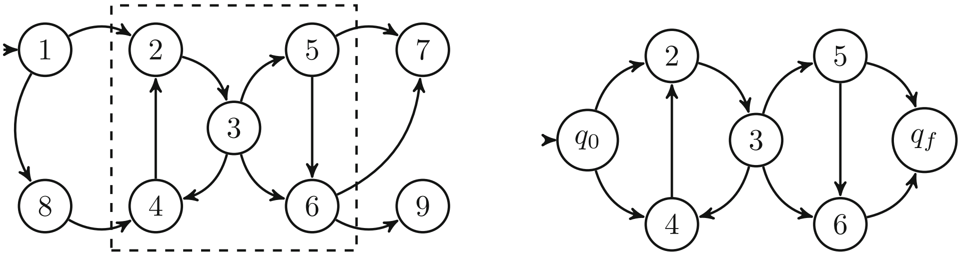
Experiment Exp2. We compared the performance of the *substitutability checker* with the LTSA model checker—a standard model checker for FLTL properties on LTS–by varying the size of the sub-components and their environments. We generated an LTS model of the environment and a complete model for the component and checked them against a property, as described in experiment Exp1. Then we extracted a sub-component by selecting a set of component states and the transitions between them, defined the pre- and post-conditions for the sub-component, and ran the substitutability checker comparing its performance with model-checking.

We describe the experiments below.

8.3.1. Experimental setup

We implemented a random LTS generator (rndLTSGen) to create LTS models with a specified number of states (#States), transition density (#Tdens) i.e., transitions per state, and number of events (#events). First, an LTS with #States states and #events events is created and then transitions are added. Starting with the initial state, we visit each state of the LTS, adding #Tdens transitions to each. Each transition has state q as source, a random state of the LTS as destination, and it is labeled with an event randomly chosen among the #events events of the LTS. The random LTS generator is used to create models for the environment and the complete component.

We also developed a partial component generator (prtCompGen) that creates a partial component with one black-box state. prtCompGen takes as input an LTS, a number of events (#events) of the LTS to be added to the interface of the black-box state, and its pre- and the post-conditions. prtCompGen randomly chooses one of the states of the LTS and marks it as black-box state and then randomly selects #events events among the LTS events and adds them to the interface of the black-box state. Finally, it sets the pre- and post-conditions of the interface to the values passed as its parameters.



(a) An LTS.

(b) The corresponding sub-component.

Fig. 12. Sub-component generation: (a) an LTS and (b) a corresponding sub-component

Table 7. Template formulae considered in the scalability evaluation

Formula	Class	Textual description
$\mathcal{K}1 = \Box(Q \rightarrow P)$	Obligation	Instance of the universality pattern specifying that $Q \rightarrow P$ holds globally.
$\mathcal{K}2 = \Diamond Q \rightarrow (\neg P \,\mathcal{U}\, Q)$	Safety	Instance of the absence pattern. It specifies that P is false before Q.
$\mathcal{K}3 = \Box(Q \rightarrow \Box(\neg P))$	Obligation	Instance of the absence pattern. It specifies that P is false after Q.

We use the formulas in Table 7 to describe pre- and post-conditions and properties of interest, where fluents Q and P are *true* when two distinct events, randomly selected among the events of the LTS, occur. Formula $\mathcal{K}1$ has been chosen since it has the same form as PR2 of our case study. The formulae $\mathcal{K}2$ and $\mathcal{K}3$ are based on the commonly used property patterns and belong to two different classes of the Manna and Pnueli hierarchy [MP90].

Finally, we implemented a sub-component generator (SUBCOMPGEN). SUBCOMPGEN takes as input an LTS and the percentage of the number of its states (%STATES) to be added. SUBCOMPGEN extracts %STATES of the states from the LTS and the transitions between them. Then it adds states q_0 and q_f as the sub-component's initial and final states, respectively, and connects, via a τ-transition, q_0 with all the states of the sub-component that have, in the LTS, at least one incoming transition from a state that was not added to the sub-component. Transitions to q_f are added similarly. To illustrate, consider the LTS in Fig. 12(a). Fig. 12(b) shows an extracted sub-component with 5 states. Transitions from q_0 to 2 and 4 are added since states 1 and 8 are not included in the sub-component and they had an outgoing transition reaching 2 and 4. Transitions from 5 and 6 to q_f are added since states 7 and 9 are not included in the sub-component and 5 and 6 had outgoing transitions reaching those states.

8.3.2. *Methodology and results*

We now describe the methodology and results of the two experiments.

Experiment EXP1. For each combination of values #EnvStates and #CompStates reported in Table 8, we generated five different configurations by changing the pre- and post-conditions of the partial component and the property of interest. We used RNDLTSGEN to generate a model of the environment $\mathcal{E}$ and a complete component $\mathcal{C}$ considering the number of states (#EnvStates for the environment and #CompStates for the complete component), the transition density and the number of events specified in Table 8. We used PRTCOMPGEN to create a partial component $\mathcal{P}$ obtained from the complete component $\mathcal{C}$ considering the number of events specified in Table 8 and by randomly selecting one among the pre- and post-conditions presented in Table 7. We also randomly selected a property ϕ from the formulae specified in Table 7.

For each configuration, we ran a classical model checker (in our case, LTSA) by checking the satisfaction of the property ϕ on the parallel composition between the environment and the complete component $\mathcal{C}$. Then we ran the well-formedness checker on the partial component $\mathcal{P}$ w.r.t. the environment $\mathcal{E}$. This procedure was run on a 2 GHz Intel Core i7, with 8 GB 1600 MHz DDR3 disk, and we recorded the average time (over the five different configurations obtained by changing pre- and post-conditions) required by the well-formedness checker (T_w) and by the model checker (T_m). Table 9 contains the average ratio among T_w and T_m for each combination of values for #EnvStates and #ContStates.

These results show that the well-formedness checker scales as well as the classical model checker does as the size of the environment, the partial component and the sub-component grows. This means that the well-formedness checker can be used in all of the scenarios where classical model checking is already being used.

Table 8. Values of the parameters considered in our experiments for the different components

Environment	**Parameter**	#EnvStates	#Tdens	#events
	Value	10, 100, 1000	10	50
Component	**Parameter**	#CompStates	#Tdens	#events
	Value	10, 50, 100, 250, 500, 750, 1000	10	50
Partial component	**Parameter**	#events	**Pre-**	**Post-**
	Value	25	$\mathcal{K}1, \mathcal{K}2, \mathcal{K}3$	$\mathcal{K}1, \mathcal{K}2, \mathcal{K}3$
Sub-component	**Parameter**	%States	**Pre-**	**Post-**
	Value	50%	$\mathcal{K}1, \mathcal{K}2, \mathcal{K}3$	$\mathcal{K}1, \mathcal{K}2, \mathcal{K}3$
Property	$\mathcal{K}1, \mathcal{K}2, \mathcal{K}3$			

Table 9. Results of experiments Exp1 and Exp2

	#CompStates													
	Exp1 : $(T_w)/(T_m)$							Exp2 : $(T_s)/(T_m)$						
#EnvStates	10	50	100	250	500	750	1000	10	50	100	250	500	750	1000
10	1.45	1.26	1.51	1.29	1.42	1.43	1.31	2.20	4.37	2.18	1.50	2.19	1.62	1.62
100	1.15	1.25	1.50	1.08	0.88	1.02	2.33	3.51	4.66	3.61	2.80	3.18	1.96	2.73
1000	1.39	1.23	0.60	1.44	4.90	1.00	2.83	13.98	8.12	3.84	2.64	2.83	2.91	2.00

Experiment Exp2. For each combination of values #EnvStates and #CompStates reported in Table 8, we again generated five configurations, each obtained by varying the pre- and post-conditions of the partial component and the property of interest. Specifically, we used rndLTSGen to generate a model of the environment $\mathcal{E}$ and a complete component $\mathcal{C}$ considering the number of states (#EnvStates for the environment and #CompStates for the complete component), the transition density and the number of events specified in Table 8, and subCompGen to create a sub-component $\mathcal{S}$ obtained from the complete component $\mathcal{C}$ considering the percentage of states specified in Table 8. We randomly chose the pre- and post-conditions of $\mathcal{S}$ and a property ϕ from the ones specified in the table.

For each configuration, we used LTSA to check whether ϕ holds on $\mathcal{E} \parallel \mathcal{C}$ and then ran the substitutability checker considering the sub-component $\mathcal{S}$ and the environment $\mathcal{E}$ on the same computer configuration as for Experiment Exp1. Table 9 contains the average ratio between the time taken by the substitutability checker (T_s) and that of the model checker (T_m) for each combination of values of #EnvStates and #ContStates.

The results show that the substitutability checker scales as well as classical model checking as the size of the environment, the partial component and the sub-component grows. This means that the substitutability checker can be used in all of the scenarios where classical model checking is already being used.

8.3.3. Discussion

The procedure employed to randomly generate models is a clear threat to construct validity. However, the transition density of the components was chosen based on the Mars Rover example. Specifically, we chose the value 10, which approximated the density of the Executive component (7.37) and the component run in parallel with it (14.96). Furthermore, the number of states was chosen such that the ratio between the sizes of the component and the sub-component was approximately the same as that of the Mars Rover example: the Executive had 96 states and the sum of the states of the LTS abstracted into black-box states (*WaitingForPlan*, *PrepareExecution* and *ExecuteTaskAction*) was 51.

The properties considered in the experiment are a threat to internal validity. However, we believe that by using properties from the patterns database, we considered properties that occur frequently, thus reducing the internal validity bias.

Considering a single black-box state is a threat to external validity. However, our goal was to evaluate how FIDDle scales with respect to the component and the environment sizes and not w.r.t. the number of black-box states and the size of the post-conditions. Considering the scalability with respect to the component and the environment sizes when multiple black-box states are present can be reduced to the problem of considering a single black-box with a more complex post-condition. Indeed, as discussed in Sect. 7, the scalabilty of the well-formedness and the substitutability checkers depend on the size of the partial component and of the post-conditions associated with its black-box states.

9. Related work

The work described in this paper proposes a complete approach—from modeling to verification and synthesis—for incremental and distributed development of correct controllers in the presence of partial information. The ability to deal with incomplete models and reason about partial information is key in our approach. It becomes necessary to support modularity and incremental design, where certain design decisions are postponed or delegated to other parties. For this reason, we consider incompleteness in every part of model development as relevant to our work, namely,

- modeling formalisms that support partial models (Sect. 9.1);
- verification techniques that permit the verification of partial models (Sect. 9.2);
- techniques to verify the pluggability of components in incomplete models or to substitute existing components (Sect. 9.3); and
- techniques to synthesize correct components in incomplete models starting from a specification of the component (Sect. 9.4).

We explore these connections below.

9.1. Modeling partiality

Since many software development processes have either a hierarchical or an iterative structure, the literature includes a variety of models for incomplete and partial models, both descriptive [RNA+04] and operational. Given the nature of the presented work and the used model, we focus only on operational models. Modal Transition Systems (MTS) [LT88], LTS$^{\uparrow}$ [GPB02], Partial Kripke Structures (PKS) [BG99], $\mathcal{X}$-Kripke Structures ($\mathcal{X}$KS) [CDEG03], and Incomplete Labeled Transition Systems (ILTS) [SS13] support the specification of incomplete concurrent systems.

MTS extend LTS by allowing incomplete transitions. MTS transitions are classified as *necessary* and *possible* (a.k.a., *maybe* or *unknown* [CBFU06]). Necessary transitions represent behavior that the system must exhibit, while possible transitions describe admissible behavior. Different operations have been defined over MTS, such as *(classical) refinement*, *observational refinement* [CBFU06], *model merging* (or logical conjunction) [LSW95]. A good overview of this area is given in [UABD+13]. PLTS differ from MTS because they express incompleteness via black-box states which are placeholders for (possibly partial) components, instead of via transitions. Moreover, the interfaces of black-box states can constrain the environment of the component that can be substituted for the black-box state. LTS$^{\uparrow}$ [GPB02] are an extension of LTS, but, unlike our approach, they do not encapsulate components in a state, but rather use the $\uparrow$ operator to make unobservable those actions in the LTS of a component that are not part of its interface.

PKS [BG99] are an extension of Kripke Structures (KS) that allow the description of incomplete models. A proposition can have a value *true*, *false* or $\perp$ (unknown) in a given state of the system. $\mathcal{X}$KS [CDEG03] extend KS to allow assigning both propositions and transitions values from a multi-valued logic. These approaches differ from PLTS where a state can be replaced (as a whole) by another (possibly incomplete) component.

ILTS [SS13] introduce the notion of *transparent states*, to handle incompleteness over states. Analogously to PLTS, ILTS use states to capture incompleteness, but they are not equipped with pre- and pos-conditions and so do not natively support distributed development.

Software product lines (SPL) [PBvDL05] provide an alternative formalism to partial systems. They allow describing and reasoning about products by considering their features. Features are (user-visible) increments in product functionalities. A software product line allows differentiating between mandatory and optional features. Mandatory features must be part of every final product. Places where different features produce different behavior are called *variation points*. Like partial models, SPL compactly represent multiple alternatives that can be generated from them, and thus partial modeling formalisms are applicable for modeling SPL. For example, in a recent work by ter Beek et al. [tBFGM16], MTS are used to describe families of products through may and must transitions which allow an efficient and compact way of modeling the notions of alternative and mutually exclusive features. The variability-aware action-based branching time modal temporal logic (v-ACTL) [TBM14] is used to reason about families of products. Our work is complementary to those, as pre- and post-conditions can be considered as a tool for constraining the features that can be obtained by specifying the behavior of the system in a black-box state.

I/O Automata [LT87, LT89, Jon94] allow modeling systems that continuously interact (receive inputs from and react to) with their environment. They are particularly useful for modeling components that operate asynchronously. Each component is modeled as an automaton whose transitions are labeled with actions. Those actions are instantaneous and classified into input, output or internal actions. While the internal and output actions of a component are controlled by the component itself, inputs are controlled by its environment and are instantaneously received by the component. Interface Automata [dAH01] are syntactically similar to the I/O Automata. However, while in I/O Automata the environment can perform an action at every state of the I/O Automata, in Interface Automata some input actions can be illegal in some of the states of a component.

Team Automata [tBK03] provide a flexible framework for modeling collaboration between components. They are similar to I/O Automata as they allow the automata to synchronize over a subset of their actions. However, different kind of synchornizations can be defined on a per-action basis. A specific notion of compatibility for Team Automata, which enables iterative and hierarchical composition, has been proposed by Carmona and Kleijn [CK13].

Finally, *Contract Automata* [FDB17] are designed to orchestrate services on a contract basis. They provide a pre-defined set of the actions, that is, "make" requests, "advertize" offers, "matching" and a pair of "complementary" request/offer. A Contract Automaton specifies the assumptions and guarantees of a service in terms of those actions. Unlike these formalisms, our formalism allows developers to explicitly identify the still to be refined parts by means of black-box states and add pre- and post-conditions to black-box states enabling iterative top-down development.

Other modeling formalisms, such as *Hierarchical State Machines* (HSM) [AY01] and *Statecharts* [Har87], support the specification of a hierarchical relation among the states of the system, while others support uncertainty [FSC12a, FSC12b, FSDSC13], i.e., they can associate incomplete parts with a set of possible replacements.

9.2. Checking partial models

A number of approaches to analyze partial models have been proposed [Hut02, CDEG03, BG99, GPB02, AY01], but none of these techniques are directly applicable to the problem considered in this paper where missing subcomponents are specified using contracts, and their development is distributed across different development teams.

In [Hut02] and [CDEG03], the authors describe the model checking problem over MTS and $\mathcal{X}$KS, respectively. In both cases, the proposed procedure first builds an under-approximation of the model, which is model checked against the property of interest. If the new model does not satisfy the property, the original model does not satisfy the property either. If this is not the case, an over-approximation is computed and then checked against the property of interest. A model checking approach for $\mathcal{X}$KS is proposed in [CDEG03]. As in the case of MTS, this problem can be reduced to a set of executions of the classical model checking algorithm. In [BG99], the authors propose a *3-valued* model checking approach, where three possible outputs can be returned: *true*, *false*, and *maybe* (or possibly). Even if able to verify incomplete systems, differently from our approach, neither these techniques are suitable for distributed development since they do not generate or assume any constraint on the incomplete components to support their development when the output of the verification depends on them.

In [AY01], the authors study the problem of checking HSM with respect to LTL properties. In this work, HSM are not considered as a formalism to model partiality, and thus the verification is assumed to be performed at the end of the development cycle when the final implementation of the model is provided. Finally, [GPB02] analyzes the assumption generation problem for LTS with an additional interface operator. This work is complementary to the one done in this paper and concerns the computation of an assumption that describes how the system model interacts with the environment. This problem can be integrated with the design phase of our approach to automatically generate post-conditions for the still-to-be-defined sub-components.

9.3. Substitutability checking

The goal of substitutability checking is to verify whether a (possibly partial) component can be plugged into a higher level structure without affecting its correctness. Thus, compositional reasoning, component substitutability and hierarchical model checking are related to this part of our work.

Compositional reasoning [dR01] reduces the verification effort by checking properties on individual components and inferring that these properties hold in the global system without explicitly creating it. For example, in the *assume-guarantee paradigm* [Jon83, AH99, Pnu85], if M guarantees ϕ and M' guarantees ψ when it is located in an environment that satisfies ϕ, then $M \mid\mid M'$ satisfies ψ. This approach works in a different direction from ours: we first guarantee that the properties of interest are satisfied in the initially defined partial model, and then check that later-provided components are suitable for plugging into the initial PLTS.

The work of Giannakopoulou et. al. [GPB02] is similar to our approach but focused on generating environmental restrictions. It addresses the assumption generation problem for LTS with an additional interface operator which describes how the model of the system interacts with its environment. In this approach, the claims to verify are also expressed as LTS. When the model of the system $\mathcal{M}$ possibly satisfies the claim Φ, the proposed algorithm computes an assumption that describes all and only environments which guarantee the satisfaction of Φ in $\mathcal{M}$. In this sense, the procedure is similar to the supervisory control problem. Another similar approach is presented in [CT15]. The work discusses how to use a previously-developed contract refinement framework in a compositional verification setting. The procedure is based on the classical structure of a deduction proof, i.e., starting from a set of axioms, the properties of the system are obtained by iteratively applying a set of inference rules.

Several techniques ensure the *substitutability between components* [ZW97, BD08]. These approaches are built upon the substitution principle of Liskov [LW94] in the context of object-oriented programming. Given a system S, they check whether the interface type[1] of a component C is a subtype of interface type of the component C' of S; this would guarantee that C can replace C' in S without causing behavior violations. The substitutability check is performed without plugging the component C into the system. This approach is analogous to the substitutability problem in the context of web services (e.g., [LFS+11]), where, once a web service W' is considered as a substitute for a web service W in a composition *Comp*, the behavioral equivalence or similarity, and input and output compatibility between W and W' in *Comp* are checked before using W' in *Comp*. The solutions proposed in literature use different formalisms, such as finite-state machines, process algebra, and Petri nets, but they all rely on similar verification approaches.

The *substitutability problem* [CCSS08] can also be defined as a check that, after substituting one or more components, any updated portion of a software system continues to provide all the services offered by its earlier counterpart, and the previously established system correctness properties remain valid for the new version of the software system. In this case, substitutability is checked only after the new components are plugged in place of the old ones.

Compositional methods, e.g., Jonsson [Jon94], aim at checking whether an I/O Automaton refines another, by omitting information about some “not interesting” communication events. These techniques are similar to the one presented in this work. However, our technique is not based on information hiding but rather aims at ensuring that certain post-conditions are guaranteed by the unspecified behavior enclosed within a black-box state.

Compositionality of Team Automata has been studied by ter Beek et al.,[tBK03] where the authors show how the relevant behavior of a composite automaton can be obtained from the behavior of its constituting automata, i.e., how, given one particular computation (behavior) of a team automaton, it is possible to extract the underlying behavior of one of its constituting component automata, and vice versa. The authors presented conditions that enable defining a set of operations that preserve this compositionality.

Ter Beek et al. [tBdV14] proposed a compositional verification technique for product lines. The compositional technique uses a driver module that coordinates between the feature and the behavioral domain model and abstracts the product behavior into a sub-component which is bisimilar to the product behavior, thereby enabling verification of local properties over a smaller behavioral model. Instead, our technique assumes a top-down development protocol in which sub-components are specified in terms of pre- and post-conditions described in FLTL.

Compositional verification can also be achieved using the Liskov substitutability Principle [LW94]. For example, Hähnle and Schaefer [HS12] studied how to extend this principle to verify software product families in a modular way. Specifically, in this work, a software product family is comprised of an original software product together with the contracts of its methods (pre- and post-conditions) and its possible variations (i.e., adding and removing methods and changing their contracts). This is different from our work in which we specifically address incremental development of behavioral specifications. A compositional verification technique based on feature verification has been proposed by Li et al. [LKF02]. The technique works by first verifying an individual feature using model checking, computing a preservation constraint that ensures the preservation of the model checking result; and proving that a feature ensures the preservation constraint of another feature. Thus, the proposed technique is similar to the classical assume-guarantee reasoning style [Jon83] which has been discussed previously.

[1] In the basic case, interface types include only information about the input and output of the component they describe, but they are also enriched with behavioral descriptions.

The ability to approximate the implementation of a sub-component by a suitable contract and to ensure correctness via substitutability is at the core of the field of *deductive software verification*. AutoProof [TFNP15, PTF15] is an auto-active verifier for object-oriented programming languages. It proves functional correctness of Eiffel programs annotated with contracts that can refer to values of variables, can include boolean operators as well as `and then` and `or else`. Dafny [Lei10] allows proving code correctness by relying on a set of high-level annotations added to the code. If the code is correct, Dafny generates a proof that the code matches the annotations. KeY [BMU15] and OpenJML [Cok11] allow verification of sequencial Java programs by checking them against properties expressed in Java Modeling Language [LBR99] (JML). KeY is an interactive program verifier, based on dynamic logic [HKT01], and is suitable for verifying complex methods as users can incrementally build proofs. In constrast, OpenJML is fully automatic, and the program annotations are used to generate verification conditions exploited by a first-order theorem prover. As a result, verification is very fast for typical boilerplate methods (getters and setters) where the correct specifications can be given directly, but the technique is less suitable for incremental specification development. VCC [DMS+09] and VeriFast [JSP+11] use SMT solvers to verify correctness properties on annotated (multi-threaded) C programs. Viper [MSS16] is an infrastructure designed to support verification of program specifications. It includes an intermediate verification language that supports encoding of different programming languages, such as Scala, Java and OpenCL [BDH15], and the Viper front-end. Why3 [FP13] is a software verification platform that includes a logical language Why, a programming language WhyML and an infrastructure for translating the logical and the programming languages to existing theorem provers that solve the verification problem. Rather than verifying software programs, FIDDle verifies a behavioral model of the system.

Finally, verification of HSM [AY01, AY98] is also related to our work since HSM support iterative refinement. However, the verification procedures for HSM assume that the HSM is fully specified and does not analyze single components in isolation.

9.4. Synthesis

Program synthesis [PR89, DBPU13] aims at computing a model of the system that satisfies the properties of interest. Moreover, synthesis can be used to generate assumptions on a system's environment to make its specification *realizable* (e.g., [LDS11]). The realizability problem is tackled in [AMT13], where the authors showed how an unrealizable specification can be refined by adding assumptions on its environment through a counter-strategy guided synthesis approach. A similar idea is proposed in [LDS11].

Synthesis approaches for controllers have been proposed. For example, in [UBC09], the authors propose a synthesis technique to construct MTS from a combination of safety properties and scenarios. The idea is that safety properties are used to synthesize a model that represents an upper bound on the behaviors of the system, i.e., they include all the possible behaviors that the system can exhibit, while scenarios are lower bounds on its behavior, i.e., they describe less behavior than what the final system should provide. In [DBPU13], the authors propose a novel synthesis technique, extending their previous work ([DBPU10, DBPU11]) and give methodological guidelines for automatically constructing event-based behavior models. The proposed approach works for an expressive subset of liveness properties, distinguishes between controlled and monitored actions, and differentiates system goals from environment assumptions.

Incomplete information and uncertain contexts are considered in [KV00] and [CPRT15]. The former considers the synthesis with incomplete information problem, which concerns the case in which each process can read only a part of the signals of the underling process (e.g., distributed programs), while the latter proposes an algorithm that allows to synthesize plans in uncertainty contexts, where an execution may result in one or more sequences of states. The goal is to compute plans which satisfy a reachability property, i.e., a condition on the final state of the execution of a plan. Finally, [AFFM06] presents Supremica, an integrated environment for verification, synthesis and simulation of discrete event systems. Supremica uses two main approaches: the first exploits modularity in order to divide the original problem into many smaller problems that together solve it, while the second uses an efficient data structure, a binary decision diagram, to symbolically represent the reachable states. [CBDU16] presents a fully automated synthesis procedure for highly non-trivial components of over 2000 states that works for special cases, by limiting the types of synthesizable goals and using heuristics. However, such cases might not be applicable in general.

Recent work has been done in the direction of *compositional* [AMT15, AMT16] and *distributed* [SUBK11] synthesis. In [AMT15], the authors studied the problem of compositional refinement of component specifications in the context of compositional reactive synthesis by considering a special case of only two components with specific constraints. In [AMT16], the synthesis problem for multi-agent systems is solved in a decomposed manner thanks to the assumption that the objective of the system is given in conjunctive form, and each conjunct of the global objective only refers to a small subset of agents in the system. Also, in [SUBK11], the authors propose a distribution algorithm for a deterministic subset of MTS. The algorithm under well-defined pre-conditions produces component MTS of a monolithic partial system behavior model without loss of information. We do not consider our approach to be an alternative to synthesis, but instead as a way to combine synthesis techniques with human design.

10. Conclusion and future work

This paper presented a verification-driven methodology, called FIDDle, to support iterative distribution of of software controllers. FIDDle enables recursive decomposition of a component into sub-components in such a way that correctness of the integrated system is ensured by construction. Development of sub-components that satisfy their specifications can then be done via independent, distributed development. FIDDle provides several types of support: modeling support for design activities performed by humans; analysis support for helping in software design; support for distributing development of unspecified parts; and support for integrating the developed sub-components.

Tool support for FIDDle was created as a Java application on top of LTSA, allowing developers to model components, iteratively develop sub-components, and perform all the checks described in this paper.

To evaluate the effectiveness of FIDDle in real cases, we abstracted parts of the K9 Mars Rover model built at NASA Ames and then simulated forward development. Our results showed that the proposed approach supports developers in the initial overall design and that the proposed analysis supports distributed development of sub-components. We also evaluated the scalability of our model-checking and substitutability procedures developed within FIDDle on randomly generated models. Our results showed that the proposed procedures scale as well as classical model-checking as the size of the environment, the partial component and the sub-component increase.

We believe that the support provided by FIDDle for iterative and incremental controller development can positively impact the way in which future software systems will be developed. One of the main limitations of the proposed framework is that pre- and post-conditions are specified through logic formulae. This can be a time-consuming and error-prone process. In our evaluation, we used property specification patterns to support us in defining pre- and post-conditions of the unspecified components. While this was effective in our evaluation, we believe that FIDDle can benefit from additional automated support for helping define and analyze pre- and post-conditions. For example, automatic extraction of pre- and post-conditions from natural language can help users with a limited logical knowledge. We plan to enhance the implementation of FIDDle and improve its performance by integrating existing symbolic model checkers, such as NuSMV [CCG+02]. We would also be very interested in further evaluating FIDDle in real scenarios, where the approach would be used throughout software development, and in conducting a user study to assess FIDDle's usability.

Acknowledgements

This work has received funding from the European Research Council under the EUs Horizon 2020 research and innovation programme (grant agreement No 731869 and No 694277).

We would like to thank the reviewers of FASE'18 for their insightful comments and Dimitra Giannakopoulou for her help with the Mars Rover case study.

References

[ABKS16] Apel S, Batory D, Kästner C, Saake G (2016) Feature-oriented software product lines. Springer, Berlin

[AFFM06] Akesson K, Fabian M, Flordal H, Malik R (2006) Supremica—An integrated environment for verification, synthesis and simulation of discrete event systems. In: Proceedings of WODES'06. IEEE, pp 384–385

[AFT08] Amalfitano D, Fasolino AR, Tramontana P (2008) Reverse engineering finite state machines from rich Internet applications. In: Proceedings of WCRE'08, pp 69–73

[AH99] Alur R, Henzinger TA (1999) Reactive modules. Formal Methods Softw Des 15(1):7–48

[AMT13] Alur R, Moarref S, Topcu U (2013) Counter-strategy guided refinement of GR(1) temporal logic specifications. In: Proceedings of FMCAD'13, IEEE, pp 26–33

[AMT15] Alur R, Moarref S, Topcu U (2015) Pattern-based refinement of assume-guarantee specifications in reactive synthesis. In: Proceedings of TACAS'15. Springer, Berlin, pp 501–516

[AMT16] Alur R, Moarref S, Topcu U (2016) Compositional synthesis of reactive controllers for multi-agent systems. In: Proceedings of CAV'16. Springer, Berlin, pp 251–269

[AY98] Alur R, Yannakakis M (1998) Model checking of hierarchical state machines. ACM SIGSOFT Softw Eng Notes 23(6):175–188

[AY01] Alur R, Yannakakis M (2001) Model checking of hierarchical state machines. ACM Trans Program Lang Syst (TOPLAS) 23(3):273–303

[BBKT04] Bensalem S, Bozga M, Krichen M, Tripakis S (2004) Testing conformance of real-time applications by automatic generation of observer. In: Proceedings of RV'04. Electronic Notes in Theoretical Computer Science, pp 23–43

[BD08] Belguidoum M, Dagnat F (2008) Formalization of component substitutability. Electron Notes Theor Comput Sci 215:75–92

[BDH15] Blom S, Darabi S, Huisman M (2015) Verification of loop parallelisations. In: International conference on fundamental approaches to software engineering. Springer, Berlin, pp 202–217

[BG99] Bruns G, Godefroid P (1999) Model checking partial state spaces with 3-valued temporal logics. In: Proceedings of CAV'99, volume 1633 of LNCS, pp 274–287

[BMS+17] Bernasconi A, Menghi C, Spoletini P, Zuck LD, Ghezzi C (2017) From model checking to a temporal proof for partial models. In: Proceeding of SEFM'17. Springer, Berlin, pp 54–69

[BMU15] Bruns D, Mostowski W, Ulbrich M (2015) Implementation-level verification of algorithms with KeY. Softw Tools Technol Transf 17(6):729–744

[Büc90] Büchi JR (1990) On a decision method in restricted second order arithmetic. In: The collected works of J. Richard Büchi. Springer, Berlin, pp 425–435

[CBDU16] Ciolek D, Braberman VA, D'Ippolito N, Uchitel S (2016) Technical report: directed controller synthesis of discrete event systems. CoRR arXiv:1605.09772

[CBFU06] Chechik M, Brunet G, Fischbein D, Uchitel S (2006) Partial behavioural models for requirements and early design. In: Proceedings of Dagstuhl seminar MMOSS, p 6351

[CCG+02] Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV 2: an opensource tool for symbolic model checking. In: Proceedings of CAV'02. Springer, Berlin, pp 359–364

[CCSS08] Chaki S, Clarke EM, Sharygina N, Sinha N (2008) Verification of evolving software via component substitutability analysis. Formal Methods Softw Des 32(3):235–266

[CDEG03] Chechik M, Devereux B, Easterbrook S, Gurfinkel A (2003) Multi-valued symbolic model-checking. ACM Trans Softw Eng Methodol (TOSEM) 12(4):371–408

[CDGV02] Calvanese D, De Giacomo G, Vardi MY (2002) Reasoning about actions and planning in LTL action theories. In: Proceedings of KR'02, volume 2, pp 593–602

[CGP99] Clarke EM (1999) Orna Grumberg, and Doron Peled. Model checking. MIT Press, Cambridge

[CGP03] Cobleigh JM, Giannakopoulou D, Păsăreanu CS (2003) Learning assumptions for compositional verification. In: Proceedings of TACAS'03. Springer, Berlin, pp 331–346

[CH01] Councill WT, Heineman GT (2001) Component-based software engineering: putting the pieces together. Addison Wesley, New York, pp 5–99

[CK13] Carmona J, Kleijn J (2013) Compatibility in a multi-component environment. Theor Comput Sci 484:1–15

[LKF02] Li HC, Krishnamurthi S, Fisler K (2002) Interfaces for modular feature verification. In: Proceedings 17th IEEE international conference on automated software engineering, pp 195–204

[Cok11] Cok DR (2011) OpenJML: JML for Java 7 by extending OpenJDK. In: NASA formal methods symposium. Springer, Berlin, pp 472–479

[CPRT15] Cimatti A, Pistore M, Roveri M, Traverso P (2015) Weak, strong and strong ycling Planning via symbolic model checking. Technical report, IRST

[CT15] Cimatti A, Tonetta S (2015) Contracts-refinement proof system for component-based embedded systems. Sci Comput Programming, 97:333–348

[DAC98] Dwyer MB, Avrunin GS, Corbett JC (1998) Property Specification patterns for finite-state verification. In: Proceedings of FMSP'98. ACM, pp 7–15

[dAH01] de Alfaro Luca, Henzinger Thomas A (2001) Interface automata. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on foundations of software engineering, ESEC/FSE-9. ACM, New York, pp 109–120

[DBPU10] D'Ippolito N, Braberman V, Piterman N, Uchitel S (2010) Synthesis of live behaviour models. In: Proceedings of SIGSOFT FSE'10. ACM, pp 77–86

[DBPU11] D'Ippolito N, Braberman V, Piterman N, Uchitel S (2011) Synthesis of live behaviour models for fallible domains. In: Proceedings of ICSE'11. IEEE, pp 211–220

[DBPU13] D'Ippolito N, Braberman V, Piterman N, Uchitel S (2013) Synthesising non-anomalous event-based controllers for liveness goals. ACM Trans Softw Eng Methodol 22

[DGV13] De Giacomo G, Vardi MY (2013) Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of IJCAI. ACM, pp 854–860

[DMS+09] Dahlweid M, Moskal M, Santen T, Tobies S, Schulte W (2009) VCC: contract-based modular verification of concurrent C. In: 2009 31st International conference on software engineering-companion volume. IEEE, pp 429–430

[dR01] de Roever W-P (2001) Concurrency verification: introduction to compositional and non-compositional methods, volume 54 of Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge

[DWDHR06] De Wulf M, Doyen L, Henzinger TA, Raskin J-F (2006) Antichains: a new algorithm for checking universality of finite automata. In: Proceedings of CAV. Springer, Berlin, pp 17–30

[FDB17] Ferrari G-L, Degano P, Basile D (2017) Automata for specifying and orchestrating service contracts. In: Logical methods in computer science, volume 12. Episciences.org

[FP13] Filliâtre J-C, Paskevich A (2013) Why3—Where programs meet provers. In: European symposium on programming, volume 7792 of Lecture Notes in Computer Science. Springer, Berlin, pp 125–128

[FSC12a] Famelis M, Salay R, Chechik M (2012) Partial models: towards modeling and reasoning with uncertainty. In: Proceedings of ICSE'12. IEEE, pp 573–583

[FSC12b] Famelis M, Salay R, Chechik M (2012) The semantics of partial model transformations. In: Proceedings of MiSE'12. IEEE, pp 64–69

[FSDSC13] Famelis M, Salay R, Di Sandro A, Chechik M (2013) Transformation of models containing uncertainty. In: Proceedings of MODELS'13. Springer, Berlin, pp 673–689

[GM03] Giannakopoulou D, Magee J (2003) Fluent model checking for event-based systems. In: Proceedings of SIGSOFT FSE'03. ACM, pp 257–266

[GPB02] Giannakopoulou D, Pasareanu CS, Barringer H (2002) Assumption generation for software component verification. In: Proceedings of ASE'02. IEEE, pp 3–12

[GPB05] Giannakopoulou D, Păsăreanu CS, Barringer H (2005) Component verification with automatically generated assumptions. J Autom Softw Eng 12(3):297–320

[GPC04] Giannakopoulou D, Pasareanu CS, Cobleigh JM (2004) Assume-guarantee verification of source code with design-level assumptions. In: Proceedings of ICSE'04. IEEE Computer Society, pp 211–220

[Har87] Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Computer Program 8(3):231–274

[HKT01] Harel D, Kozen D, Tiuryn J (2001) Dynamic logic. In: Handbook of philosophical logic. Springer, Berlin, pp 99–217

[HS12] Hähnle R, Schaefer I (2012) A Liskov principle for delta-oriented programming. In: Tiziana M, Bernhard S (eds) Leveraging applications of formal methods, verification and validation. Technologies for mastering change. Springer, Berlin, pp 32–46

[Hut02] Huth M (2002) Model checking modal transition systems using kripke structures. In: Proceedings of VMCAI'02, volume 2294 of LNCS, pp 302–316

[Jon83] Jones CB (1983) Tentative steps toward a development method for interfering programs. ACM Trans Programm Lang Syst (TOPLAS) 5(4):596–619

[Jon94] Jonsson B (1994) Compositional specification and verification of distributed systems. ACM Trans Program Lang Syst 16(2):259–303

[JSP+11] Jacobs B, Smans J, Philippaerts P, Vogels F, Penninckx W, Piessens F (2011) VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: NASA formal methods symposium. Springer, New York, pp 41–55

[Kel76] Keller RM (1976) Formal verification of parallel programs. Commun ACM 19(7):371–384

[KV00] Kupferman O, Vardi M (2000) Synthesis with incomplete information. In: Advances in temporal logic. Springer, Berlin, pp 109–127

[LBR99] Leavens GT, Baker AL, Ruby C (1999) JML: a notation for detailed design. In: Behavioral specifications of businesses and systems. Springer, Berlin, pp 175–188

[LDS11] Li W, Dworkin L, Seshia SA (2011) Mining assumptions for synthesis. In: Proceedings of ACM/IEEE MEMCODE'11, pp 43–50

[Lei10] Leino KRM (2010) Dafny: an automatic program verifier for functional correctness. In: International conference on logic for programming artificial intelligence and reasoning. Springer, Berlin, pp 348–370

[Lev87] Levy LS (1987) Taming the tiger: software engineering and software economics. Springer Books on Professional Computing Series. Springer-Verlag, Berlin

[LFS+11] Li X, Fan Y, Sheng QZ, Maamar Z, Zhu H (2011) A petri net approach to analyzing behavioral compatibility and similarity of web services. IEEE Trans Syst Man Cybern A Syst Hum 41(3):510–521

[LMP08] Lorenzoli D, Mariani L, Pezzè M (2008) Automatic generation of software behavioral models. In: Proceedings of ICSE'08, pp 501–510

[LSW95] Larsen KG, Steffen B, Weise C (1995) A constraint oriented proof methodology based on modal transition systems. In: BRICS notes. Springer, Berlin, pp 17–40

[LT87] Lynch N, Tuttle MR (1987) Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the sixth annual ACM symposium on principles of distributed computing. ACM, pp 137–151

[LT88] Larsen KG, Thomsen B (1988) A modal process logic. In Proceedings of LICS'88, IEEE, pp 203–210

[LT89] Lynch NA, Tuttle MR (1989) An introduction to input/output automata. CWI Q 2:219–246

[LW94] Liskov BH, Wing JM (1994) A behavioral notion of subtyping. Trans Program Lang Syst (TOPLAS) 16(6):1811–1841

[MGPT18a] Menghi C, Garcia S, Pelliccione P, Tumova J (2018) Multi-robot LTL planning under uncertainty. In: Proceedings of FM'18

[MGPT18b] Menghi C, García S, Pelliccione P, Tumova J (2018) Towards multi-robot applications planning under uncertainty. In: Proceedings of ICSE'18, companion proceedings

[MK99] Magee J, Kramer J (1999) State models and Java programs. Wiley, Hoboken

[MP90] Manna Z, Pnueli A (1990) A hierarchy of temporal properties. In: Proceedings of PODC'90. ACM, pp 377–410

[MS99] Miller R, Shanahan M (1999) The event calculus in classical logic—alternative axiomatizations. Elect Trans AI 4

[MSCG18] Menghi C, Spoletini P, Chechik M, Ghezzi C (2018) Supporting verification-driven incremental distributed design of components. In: Proceedings of FASE'18. Springer, Berlin

[MSG16] Menghi C, Spoletini P, Ghezzi C (2016) Dealing with incompleteness in automata-based model checking. In: Proceedings of FM'16. Springer, Berlin, pp 531–550

[MSG17] Menghi C, Spoletini P, Ghezzi C (2017) Integrating goal model analysis with iterative design. In: Proceedings of REFSQ'17. Springer, Berlin, pp 112–128

[MSS16] Müller P, Schwerhoff M, Summers AJ (2016) Viper: a verification infrastructure for permission-based reasoning. In: International conference on verification, model checking, and abstract interpretation. Springer, Berlin, pp 41–62

[Par72a] Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058

[Par72b] Parnas DL (1972) A technique for software module specification with examples. Commun ACM 15(5):330–336

[PBB+04] Pistore M, Barbon F, Bertoli P, Shaparau D, Traverso P (2004) Planning and monitoring Web service composition. In: Proceedings of AIMSA'04, Springer, Berlin, pp 106–115

[PBKS07] Pretschner A, Broy M, Kruger IH, Stauner T (2007) Software engineering for automotive systems: a roadmap. In: Proceedings of FOSE'07, IEEE Computer Society, pp 55–71

[PBvDL05] Pohl K, Böckle G, van Der Linden FJ (2005) Software product line engineering: foundations, principles and techniques. Springer, Berlin

[Pnu85] Pnueli A (1985) In transition from global to modular temporal reasoning about programs. In: Krzysztof RA (ed) Logics and models of concurrent systems. Springer, New York, pp 123–144

[PR89] Pnueli A, Rosner R (1989) On the synthesis of a reactive module. In: Proceedings of POPL'89. ACM, pp 179–190

[PTF15] Polikarpova N, Tschannen J, Furia CA (2015) A fully verified container library. In: Formal methods (FM), Lecture Notes in Computer Science. Springer, Berlin

[RNA+04] Díaz RRP, Nores ML, Pazos AJJ, Vilas AF, Duque JG, Solla AG, Martínez BB, Cabrer MR (2004) Supporting software variability by reusing generic incomplete models at the requirements specification stage. In: Jan B, Charles K (eds) Software reuse: methods, techniques, and tools. Springer, Berlin

[San95] Sandewall E (1995) Features and fluents (vol 1): the representation of knowledge about dynamical systems. Oxford University Press, Oxford

[SFG+12] Saadatpanah P, Famelis M, Gorzny J, Robinson N, Chechik M, Salay R (2012) Comparing the effectiveness of reasoning formalisms for partial models. In: Proceedings of MoDeVVA'12, ACM, pp 41–46

[SL08] Solar-Lezama A (2008) Program synthesis by sketching. PhD thesis, University of California, Berkeley

[SL13] Solar-Lezama A (2013) Program sketching. STTT 15(5):475–495

[Sof04] Software Measurement Services Ltd (2004) "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

[SS13] Sharifloo AM, Spoletini P (2013) Lover: light-weight formal verification of adaptive systems at run time. In: Formal aspects of component software. Springer, Berlin, pp 170–187

[SUBK11] Sibay GE, Uchitel S, Braberman V, Kramer J (2011) Distribution of modal transition systems. In: Proceedings of FM'11, pp 403–417

[tBdV14] ter Beek MH, de Vink EP (2014) Towards modular verification of software product lines with mCRL2. In: Tiziana M, Bernhard S (eds) Leveraging applications of formal methods, verification and validation. Technologies for Mastering Change. Springer, Berlin, pp 368–385

[tBFGM16] ter Beek MH, Fantechi A, Gnesi S, Mazzanti F (2016) Modelling and analysing variability in product families: model checking of modal transition systems with variability constraints. J Log Algeb Methods Program 85(2):287–315

[tBK03] ter Beek MH, Kleijn J (2003) Team automata satisfying compositionality. In: Formal methods (FM). Springer, Berlin, pp 381–400

[TBM14] Ter Beek MH, Mazzanti F (2014) VMC: recent advances and challenges ahead. In: International software product line conference: companion volume for workshops, demonstrations and tools-volume 2. ACM, pp 70–77

[tBRdV16] ter Beek MH, Reniers MA, de Vink EP (2016) Supervisory controller synthesis for product lines using CIF 3. In: Tiziana M, Bernhard S (eds) Leveraging applications of formal methods, verification and validation: foundational techniques. Springer, Berlin, pp 856–873

[TFNP15] Tschannen J, Furia CA, Nordio M, Polikarpova N (2015) AutoProof: auto-active functional verification of object-oriented programs. In: International conference on tools and algorithms for the construction and analysis of systems. Lecture Notes in Computer Science. Springer, Berlin, pp 566–580

[TV05] Tabakov D, Vardi MY (2005) Experimental evaluation of classical automata constructions. In: Proceedings of LPAR. Springer, Berlin, pp 396–411

[TV07] Tabakov D, Vardi MY (2007) Model checking Buchi specifications. In: Proceedings of LATA, pp 565–576

[UABD+13] Uchitel S, Alrajeh D, Ben-David S, Braberman V, Chechik M, De Caso G, D'Ippolito N, Fischbein D, Garbervetsky D, Kramer J, Russo A, German SE (2013) Supporting incremental behaviour model elaboration. Comput Sci-Res Dev 28(4):279–293

[UBC09] Uchitel S, Brunet G, Chechik M (2009) Synthesis of partial behavior models from properties and scenarios. IEEE Trans Softw Eng 35(3):384–406

[vBFH+14] van Beek DA, Fokkink WJ, Hendriks D, Hofkamp A, Markovski J, van de Mortel-Fronczak JM, Reniers MA (2014) CIF 3: model-based engineering of supervisory controllers. In: Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 575–580

[VW94] Vardi MY, Wolper P (1994) Reasoning about infinite computations. J Inf Comput 115(1):1–37

[YPA06] Yuan J, Pixley C, Aziz A (2006) Constraint-based verification. Springer, Berlin

[ZW97] Zaremski AM, Wing JM (1997) Specification matching of software components. ACM Trans Softw Eng Methodol 6(4):333–369

Received 4 October 2018

Accepted in revised form 24 May 2019 by Alessandra Russo, Andy Schuerr, and Heike Wehrheim