Available online at www.sciencedirect.com
ScienceDirect
Journal of Applied Logic 5 (2007) 333–355
JOURNAL OF APPLIED LOGIC
www.elsevier.com/locate/jal

Precise specification matching for adaptive reuse in embedded systems

Hai-Feng Guo [a,*], Miao Liu [a], Partha S. Roop [b], C.R. Ramakrishnan [c], I.V. Ramakrishnan [c]

[a] *Department of Computer Science, University of Nebraska at Omaha, Omaha, Nebraska 68182, USA*
[b] *Department of Electrical and Electronic Engineering, University of Auckland, Private Bag 92019, Auckland, New Zealand*
[c] *Department of Computer Science, Stony Brook University, Stony Brook, NY 11794, USA*

Received 21 July 2004; accepted 12 December 2005

Available online 4 May 2006

Abstract

Specification matching is a key to reuse of components in embedded systems. Existing specification matching techniques for embedded systems are designed to match reactive behaviors using adaptive techniques to dynamically alter behaviors. However, correct specification matching demands both *behavioral matching* (that checks component adaptability) and *functional matching* (that ensures that proper functionality is reused). While approaches for behavioral matching exist, combined functional and behavioral matching during component reuse in embedded systems is lacking. This paper presents a precise specification matching, including both behavioral and functional matching. We introduce *attributed labeled transition systems* (*ALTS*) to formally specify component behavior and functionalities. Given ALTS of a new specification (a function $\mathcal{F}$) and an existing component (a device $\mathcal{D}$), a new refinement relation from $\mathcal{F}$ to $\mathcal{D}$, called an S-matching relation, is proposed for precise specification matching. The existence of an S-matching relation is also shown to be a necessary and sufficient condition for the existence of a correct *adapter* to adapt $\mathcal{D}$ to match $\mathcal{F}$ both behaviorally and functionally. Automated component adaptation is facilitated by a matching tool implemented in a tabled logic programming environment, which provides distinct advantages for rapid implementation. Practical examples are given to illustrate how the concrete adapter is derived automatically from specification matching.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Component reuse; Specification matching; Formal methods; Tabled resolution; Logic programming; Justification

1. Introduction

Embedded Systems are ubiquitous digital systems that continuously interact with their immediate environment (also known as *reactive systems*). With the widespread demand for embedded systems in applications ranging from home appliances to complex controllers for aircraft and power plants, there is an increasing need for rapid implementation and validation to tackle the short time to market for these products. Component reuse [13] (also known as IP (intellectual property) reuse) is being proposed as an alternative design paradigm for these systems. Moreover, as embedded

* Corresponding author.
E-mail addresses: haifengguo@mail.unomaha.edu (H.-F. Guo), miaoliu@mail.unomaha.edu (M. Liu), p.roop@auckland.ac.nz (P.S. Roop), cram@cs.sunysb.edu (C.R. Ramakrishnan), ram@cs.sunysb.edu (I.V. Ramakrishnan).

1570-8683/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.jal.2005.12.016

systems are often safety critical, formal techniques [8,10,22,34] are increasingly applied to the design and validation of these systems.

There are usually two main problems involved in component reuse: *component retrieval* and *component adaptation*. The former is used to perform reusable component identification over a library of prefabricated components; and the latter can adapt the retrieved component to a new function. To facilitate plug-and-play [20] component reuse both retrieval and adaptation need to be automated. This paper addresses the component adaptation by developing a formal approach for the problem.

Many studies have been carried out to seek different formal methods to handle component retrieval and adaptation. *Specification matching* [2,9,12,24,27,36] for *transformational* software components has been developed as a foundation for recognizing and retrieving a reusable component that fits the objectives of a new design. In this setting, matching criteria are defined based on relationship between pre- and post-conditions of a query and the corresponding component being matched. Such an approach to matching, while useful for conventional software, is inappropriate for reactive systems such as embedded systems as the matching requires transformation of reactive behaviors using some dynamic techniques.

Component adaptation techniques, such as superimposition [5], dynamic component adaptation [19] and behavioral specification [31], specify behavioral matching similarly, but without a validation proof. Recently, Roop [28] proposed a formal method, called *forced simulation*, which checks whether a given programmable device can be adapted to match a given behavioral specification. When conventional refinement [1,17] fails, forced simulation may be used for checking *adaptive refinement*. If such an adaptation is possible, a device driver (referred to as an adapter) is generated to perform the adaptation. However, the proposed matching cannot guarantee that the matched components, after adaptation, have the same functionality as desired in the requirements.

In this paper, we focus on the problem of adapting hardware or software components for embedded applications (hereafter referred to simply as *devices*) for *precise specification matching* that includes behavioral and functional matching. To automatically reuse devices during component synthesis, programmable devices are normally indexed from a library. Subsequent to indexing, matching is essential to identify the exact device $\mathcal{D}$ that is both behaviorally and functionally adaptable to the design function $\mathcal{F}$ (hereafter referred to as *function*). Behavioral matching is used to check adaptability, while functional matching is to ensure the proper functionality.

Specification matching can be a tedious and time consuming activity. It is even more severe with programmable and parameterizable devices which may not match the design function $\mathcal{F}$ exactly, but may be programmed via a *device adapter* (referred to as an *adapter* $\mathcal{I}$). This paper proposes a precise specification matching technique suitable for fast logic programming based implementation. Our approach may be used for automatically deciding whether a given $\mathcal{D}$ matches $\mathcal{F}$. The matching tool, when successful, automatically generates an adapter $\mathcal{I}$ that can adapt $\mathcal{D}$ to $\mathcal{F}$. On the other hand, when the specification matching fails, the tool can justify the failure by showing partially matched specification and highlighting critical mismatching states. This information may be useful for suitably modifying $\mathcal{D}$ if it is available in the form of a *soft core*[1] [28].

We introduce an *attributed labeled transition system* (*ALTS*) to formally specify both the behaviors and functionalities of a device as a whole. The proposed approach is suitable for modeling both hardware and software components used in embedded systems. Behaviors are represented by a finite set of states and transitions defined over states, and functionalities are defined by the relations between input and output behaviors. Therefore, specification matching is to check whether a given ALTS $\mathcal{M}_1$, representing a programmable device, can be adapted to match another given ALTS $\mathcal{M}_2$, representing a new device to design. This specification matching is different from forced simulation [28] in the sense that our new approach includes a formal specification for high level functional matching, which leads to a more precise component adaptation as a result.

This paper also demonstrates the advantages of tabled logic programming system such as XSB [35] to implement a formal specification matching tool. XSB has already been used for building efficient model checkers [26] and bisimulation checkers [4]. More importantly, the XSB system provides a *justifier* [23], which essentially constructs concise evidence or debugging information to support the results of query evaluation. This justifier, with tabled logic programming, can be easily extended to provide an attractive platform for encoding computational problems, such as component matching and adaptation, in the specification and verification of systems.

[1] A *soft core* is typically a block of digital logic defined at higher level (e.g., in terms of states and transitions) than a *hard core*, which is at the gate level.

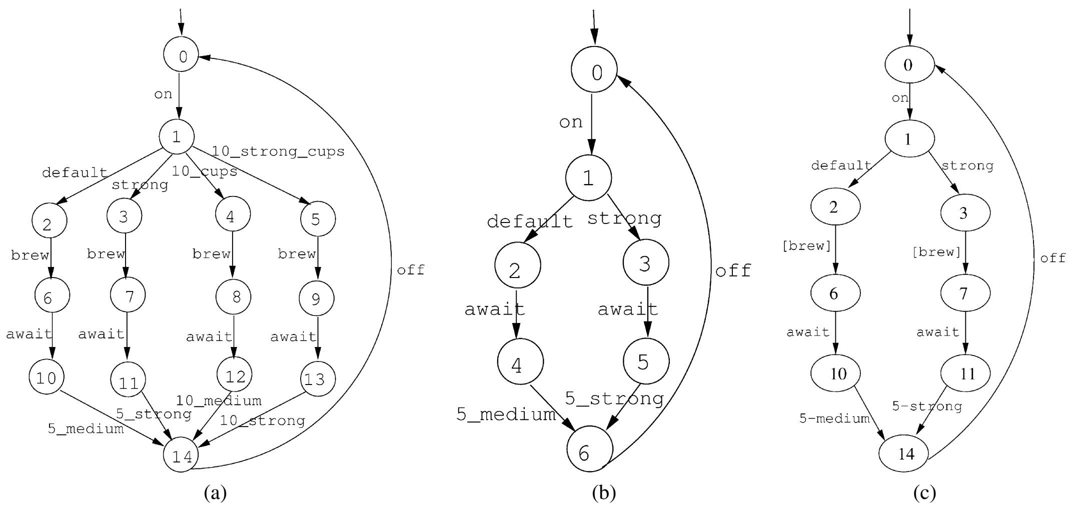


Fig. 1. (a) A coffee brewer device $\mathcal{D}$. (b) Specification of a simple coffee brewer $\mathcal{F}$. (c) A device driver (an adapter $\mathcal{I}$).

The main contributions of this paper are: (1) A new finite state model for the precise specification of programmable components is developed (for embedded systems). The model is capable of representing both the behavioral and functional aspects of a component. (2) A formal specification matching scheme to check the adaptability of a component ($\mathcal{D}$) to match a new specification ($\mathcal{F}$) is proposed. We have developed an *adaptive refinement* from $\mathcal{F}$ to $\mathcal{D}$, called S-matching relation, for the automatic detection of component adaptability. (3) The existence of an S-matching relation between $\mathcal{F}$ and $\mathcal{D}$ is shown to be a necessary and sufficient condition for precise specification matching through an adapter. (4) A fully automated procedure to generate device adapters has been implemented and the approach is validated by reusing several real embedded system components.

The rest of the paper is organized as follows: Section 2 introduces behavioral matching method and its drawbacks; Section 3 presents a formal specification matching method which performs both behavioral and functional checking. Section 4 addresses logic programming based implementation of our specification matching (S-matching) tool. Coffee brewer adaptation problems will be used to illustrate the details of precise specification and automated adapter generation. The same example will also highlight the problems with existing behavioral matching techniques. More examples including both hardware and software components are developed in Section 5 to test our S-matching tool. Section 6 presents related research in formal verification and discrete event control. Section 7 presents the conclusions of this work.

2. Behavioral matching

This section presents a behavioral matching framework called *forced simulation* [28]. We use a coffee brewer example, developed by Roop [28], to illustrate how to perform the simulation.

2.1. A coffee brewer adaptation problem

A general coffee brewer is presented in Fig. 1(a) as a programmable device $\mathcal{D}$, allowing the user to brew 5 or 10 cups of coffee with brew strength switch. The user may select brew strength and number of cups in any combination, where `default` means 5 cups of coffee with medium strength. A simple coffee brewer is also shown in Fig. 1(b) as a design function $\mathcal{F}$, demanding a brewer that provides only five cups of coffee. Now the question is how to reuse the general coffee brewer $\mathcal{D}$ to realize the functionality of $\mathcal{F}$.

In order for $\mathcal{D}$ to simulate $\mathcal{F}$, a device driver (an *adapter*[2] $\mathcal{I}$ shown in Fig. 1(c)) is needed which can dynamically adapt $\mathcal{D}$ in the following way, starting from the state 0:

[2] The term *adapter* here is same as the term *interface* used in [28].

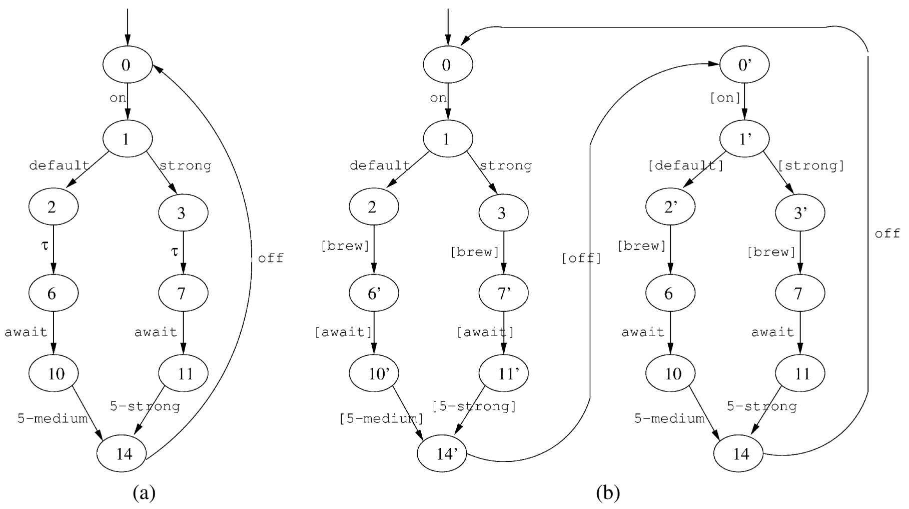


Fig. 2. (a) $\mathcal{I}//\mathcal{D}$. (b) A valid but bad adapter $\mathcal{I}$.

(1) When $\mathcal{D}$ is in state 0, the adapter $\mathcal{I}$ can enable the transition $0 \to 1$ with action on since there is a matched transition in $\mathcal{F}$. Such an action is called a *matched* action, which is shown in $\mathcal{I}$ as a normal action.
(2) When $\mathcal{D}$ is in state 1, $\mathcal{I}$ must disable the two actions 10_cups and 10_strong_cups since they are not present in $\mathcal{F}$. Such actions are named as *disabled* actions, which simply are not shown in the adapter $\mathcal{I}$.
(3) When $\mathcal{D}$ is in state 2 (or 3), $\mathcal{I}$ has to forcefully perform the action brew to reach its next state 6 (or 7), so that further simulation can be continued. Such an action is called a *forced* action. In the adapter, forcing signals are enclosed in [ ] to clearly distinguish them from other signals.
(4) The adapter $\mathcal{I}$ must perform such *forced* and *matched* actions until the desired behavior is realized.

This example raises the following additional questions: (1) Given an arbitrary pair of $\mathcal{F}$ and $\mathcal{D}$, how do we decide whether an adapter exists or not? (2) If such an adapter exists, how can we automatically derive it? (3) Given an adapter for a known pair of $\mathcal{F}$ and $\mathcal{D}$, how can we ensure that $\mathcal{D}$ implements all behaviors in $\mathcal{F}$? In other words, how do we know that the adapter is correct?

2.2. *Forced simulation*

Forced simulation was proposed as a formal method to handle the questions in the previous section. Given a pair of $\mathcal{F}$ and $\mathcal{D}$, the main idea is to build an adapter $\mathcal{I}$ such that the composition of $\mathcal{I}$ and $\mathcal{D}$ exhibits the same behavior as $\mathcal{F}$. The processes $\mathcal{F}$, $\mathcal{D}$, and $\mathcal{I}$ are modeled as *labeled transition systems*[3] [21].

The composition of $\mathcal{I}$ and $\mathcal{D}$ is defined by introducing a new parallel operator // [29] which combines a forced move in $\mathcal{I}$ with a corresponding transition in $\mathcal{D}$ to generate an unobservable τ move in $\mathcal{I}//\mathcal{D}$. Similarly, the // operator combines a matched move in $\mathcal{I}$ with an identical move in $\mathcal{D}$ resulting in an observable external move in $\mathcal{I}//\mathcal{D}$. For the coffee brewer example, the $\mathcal{I}//\mathcal{D}$ is shown in Fig. 2(a).

For $\mathcal{D}$ to match $\mathcal{F}$, $\mathcal{I}//\mathcal{D}$ must be behaviorally equivalent to $\mathcal{F}$. This is checked by using Milner's weak bisimulation [21] (also known as *observational equivalence*). Intuitively, two processes are weakly bisimilar if their behaviors cannot be distinguished by an external observer. Note that, for the coffee brewer example, the behavior of $\mathcal{F}$ in Fig. 1(b) is observationally equivalent to $\mathcal{I}//\mathcal{D}$ in Fig. 2(a) (the only difference between $\mathcal{I}//\mathcal{D}$ and $\mathcal{F}$ are some internal τ steps of $\mathcal{I}//\mathcal{D}$ which are unobservable).

The matching algorithm based on forced simulation overlooked an important fact that forced actions are different from unobservable τ steps in the sense that forced actions contain behaviors which may affect the final result. For

[3] A process is described by a labeled transition system (LTS) which is a tuple of the form $\langle S, s_0, \Sigma, \to\rangle$, where: S is a finite set of states, $s_0 \in S$ is a unique start state, Σ is a finite set of events or signals and $\to \subseteq S \times \Sigma \times S$ is the transition relation.

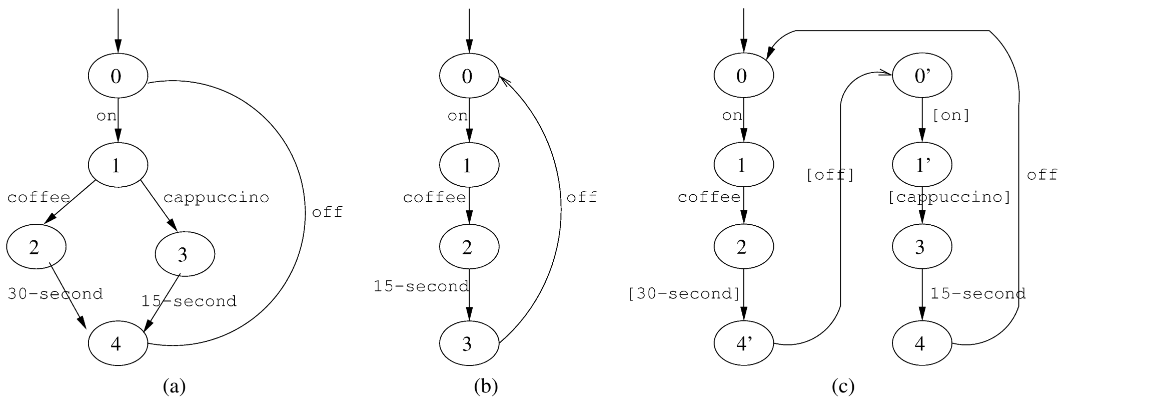


Fig. 3. (a) A coffee/cappuccino brewer device $\mathcal{D}$. (b) Specification of a coffee-only brewer $\mathcal{F}$. (c) A bad device adapter $\mathcal{I}$.

instance, Fig. 2(b) is another valid adapter to the coffee brewer adaptation problem, since if we replace all the forced actions by τ transitions, the resulting LTS is observationally equivalent to $\mathcal{F}$. However, the adapted coffee brewer with adapter $\mathcal{I}$ in Fig. 2(b) will produce ten cups of coffee every cycle from action `on` to `off`, which is obviously not the expected behavior.

Even worse, Fig. 3 shows another adaptation example, where the adapter $\mathcal{I}$, generated using forced simulation, behaves completely different from the expected simple coffee brewer with 15 seconds per cup. In fact, it is not possible to adapt the given device to the function.

It may be argued that we can simply put a constraint to avoid loops during simulation. However, the self-loop is sometimes required in simulation, for example, in order to simulate a new simple brewer coffee machine producing two cups of coffee every cycle using the coffee/cappuccino brewer device (shown in Fig. 3(a)), the adapter has to involve two cycles of the original brewer device (Fig. 3(a)).

2.3. Problems in behavioral matching

Forced simulation is actually used to produce a standard adapter for component reuse at the behavioral level. We denote the formal description at the behavioral level as *behavioral specification*. And the formalized matching defined on behavioral specification is called *behavioral matching*. Generally, behavioral matching has the following problems:

- Behavioral specification usually describes components in a syntactic way, encapsulating the functional behaviors as a black box. However, component reuse is essentially a matter of reusing the correct functionality, while preserving behavior.
- Behavioral matching using weak bisimulation loses accuracy by replacing forced actions with unobservable τ transitions. Such an action, ignores the semantics of any functional operations that are performed on a transition being forced by just hiding this in the composite system. This may lead to incorrect reuse of functionality as illustrated by the adapter in Fig. 3(c). In this case, the adapter actually forced a 30-*second* transition as well as a *cappuccino* transition while also forcing the *off*, *on* switches in between. As a result, the user waits 45 minuets to get coffee while he also gets a cappuccino before getting the coffee he requested. This was not the intended behavior of the coffee machine in Fig. 3(b).
- The adapter produced by the behavioral matching can only be used to adapt the device to be reused at a physical or mechanical level instead of a functional level. Consequently, properties of the adapter have to be verified separately in a different way to ensure that its functionality correct.

3. Precise specification matching

In this section, we firstly introduce an attributed labeled transition system (ALTS) to formally specify both the behaviors and functionalities of an embedded device. Then, we present a mathematical relation, called *specification matching* (S-matching), between two given ALTSs. We show that existence of an S-matching relation between the

ALTS specifications of a programmable device $\mathcal{D}$ and a new design function $\mathcal{F}$ is a necessary and sufficient condition for adaptability from $\mathcal{D}$ to $\mathcal{F}$. Finally, we explain how device adaptation can be achieved in our specification matching framework.

3.1. Attributed labeled transition systems

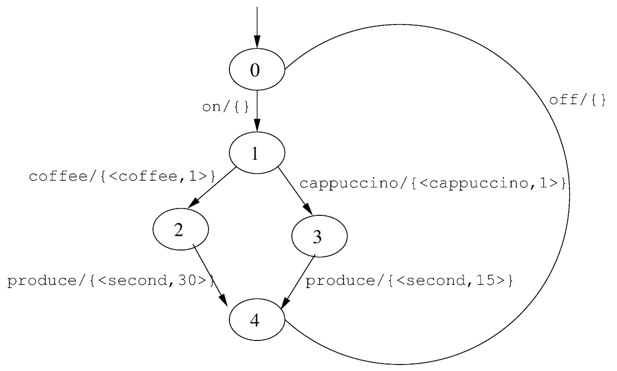
We present a new automaton notation called *attributed labeled transition system* (ALTS) for precisely specifying components used in embedded systems to capture both behavioral and functional aspects. Models like labeled transition systems (LTS) used in [28] cannot distinguish between inputs and outputs. Later IP matching algorithms [30] used codesign finite state machine (CFSM) [3] like FSM models to alleviate this problem. Each transition in a CFSM is triggered based on some inputs and may produce some outputs. ALTSs are similar in that they distinguish between inputs and outputs; the outputs are specifically modeled as attribute pairs to enable functional matching of quantitative attributes of the system.

Attributes, in the form of a pair $\langle A, V\rangle$, are used to describe functional behaviors of processes, including timings, number of products, or any other quantitative measures of a system, where A is an attribute name and V is the attribute value. Attribute values are atomic, such as integers, or reals. For simplicity, we only consider the numeral types and arithmetic operations in this paper. However, the attributes can be easily extended to support other types of values, such as strings or other compound data.

Definition 1 *(ALTS).* A process is described by an *attributed labeled transition system* (*ALTS*) which is a tuple of the form $\langle Q, \Sigma, \delta, q_0, \Gamma, \lambda\rangle$, where:

(1) Q is a finite set of states, $q_0 \in Q$ is a unique start state;
(2) Σ is a finite set of events or input signals;
(3) $\delta : Q \times \Sigma \to Q_\perp$ is a transition partial function that takes as arguments a state and an input signal and returns a state or a special symbol $\perp$ denoting no such transition, where $Q_\perp = Q \cup \{\perp\}$;
(4) Γ is a finite set of attribute pairs;
(5) $\lambda : Q \times \Sigma \to \Gamma_\perp^*$, denoted as an *output* partial function, takes as arguments a state and an input signal and returns a list of attribute pairs or a special symbol $\perp$, where $\delta(q, a) = \perp$ if and only if $\lambda(q, a) = \perp$ for any state q and input signal a.

In our informal transition diagram representation of ALTS, δ and λ were represented by arcs between states and the labels on the arcs. If q is a state, and a is an input signal, then $\delta(q, a)$ is the state p such that there is an arc labeled $a/\lambda(q, a)$ from q to p, usually written as $q \xrightarrow{a/\lambda(q,a)} p$, where $\lambda(q, a)$ is a list of attributes paired with $\{\}$. Because of this, a is usually called *input labels*, and $\lambda(q, a)$ called *output labels*. We use `functions` instead of `relations` in the above definition, because it is assumed that all processes are deterministic.



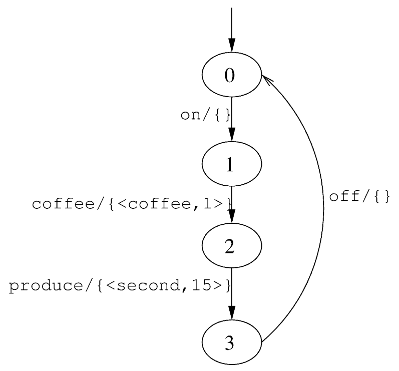


Fig. 4. (a) An ALTS diagram of coffee/cappuccino brewer device $\mathcal{D}$. (b) An ALTS specification of a coffee-only brewer $\mathcal{F}$.

The motivations for using attributes in ALTSs to model embedded systems are as follows: (1) by using the attribute pairs as output labels, the transitions are simpler to specify in the sense that they trigger based on abstract actions; (2) the attribute pairs are particularly used during the matching process to check correct functionality.

An attributed labeled transition system is a generalization of a labeled transition system with an output function (similar to Mealy machines [33]). It uses a combination of the transition function and the output function to precisely describe the behaviors of hardware/software components. Two example ALTS diagrams are illustrated in Fig. 4(a) and (b), which correspond to the coffee/cappuccino brewer device $\mathcal{D}$ and a design specification of a coffee-only brewer $\mathcal{F}$ respectively. Behaviors, such as turning on, choosing coffee or cappuccino, producing the selected product, or turning off, are represented by the input labels. On the other hand, functional properties, such as the quantity of produced goods and temporal information, are indicated by the output labels.

3.2. Specification matching

To precisely describe the behavior of an ALTS, we must extend the transition function δ and the output function λ to apply to a signal sequence rather than a single input signal. The function δ can be extended to a function $\hat{\delta}$ mapping $Q \times \Sigma^*$ to $Q_\perp$:

1. $\hat{\delta}(q,\epsilon) = q$, where ϵ is an empty sequence, and
2. for any input signal sequence ω and an input label a,

$$\hat{\delta}(q,\omega a) = \begin{cases} \perp & \text{if } \hat{\delta}(q,\omega) = \perp; \\ \delta(\hat{\delta}(q,\omega), a) & \text{otherwise.} \end{cases}$$

The intention is that $\hat{\delta}(q,\omega)$ is the state in which the ALTS will be after reading the input signal sequence ω starting in state q. Similarly, the function λ can be extended to a function $\hat{\lambda}$ mapping $Q \times \Sigma^*$ to $\Gamma_\perp^*$, which can be formally defined as:

1. $\hat{\lambda}(q,\epsilon) = \emptyset$ (or $\{\}$), where ϵ is an empty sequence, and
2. for any input signal sequence ω and an input label a,

$$\hat{\lambda}(q,\omega a) = \begin{cases} \perp & \text{if } \hat{\delta}(q,\omega a) = \perp; \\ \hat{\lambda}(q,\omega) \uplus \lambda(\hat{\delta}(q,\omega), a) & \text{otherwise,} \end{cases}$$

where the binary operator $\uplus$ is to union two sets of attribute pairs with the values for the same attribute added together:

$$\begin{aligned} S_1 \uplus S_2 = &\big\{\langle a, v_1 + v_2\rangle \mid \langle a, v_1\rangle \in S_1 \wedge \langle a, v_2\rangle \in S_2\big\} \\ &\cup\big\{\langle a, v\rangle \mid \langle a, v\rangle \in S_1 \wedge \langle a, \_\rangle \notin S_2\big\} \\ &\cup\big\{\langle a, v\rangle \mid \langle a, \_\rangle \notin S_1 \wedge \langle a, v\rangle \in S_2\big\}, \end{aligned}$$

where S_1 and S_2 are sets of attribute pairs, and the notation '_' represents any numeral value.

An attribute-based constraint is a simple inequality on an output function λ or its extended one $\hat{\lambda}$. For example: $\hat{\lambda}(p,\omega_1) = \hat{\lambda}(q,\omega_2)$ means that the signal sequence ω_1 starting from state p has the same functional behaviors as the signal sequence ω_2 starting from state q.

Definition 2 *(Specification matching (S-matching)).* Given two ALTSs: $\mathcal{F} = \langle Q_f, \Sigma_f, \delta_f, q_{f_0}, \Gamma_f, \lambda_f\rangle$, $\mathcal{D} = \langle Q_d, \Sigma_d, \delta_d, q_{d_0}, \Gamma_d, \lambda_d\rangle$, a relation $R \subseteq Q_d \times Q_f$ is a *specification matching* relation (in short, an *S-matching* relation where $q_d R q_f$ is a shorthand for $(q_d, q_f) \in R$) provided that:

(1) $\forall q_f \in Q_f, \exists q_d \in Q_d$ s.t. $q_d R q_f$; and
(2) $\forall (q_d, q_f) \in Q_d \times Q_f$, whenever $q_d R q_f$, it holds that $\forall a \in \Sigma_f$ s.t. $q'_f = \delta_f(q_f, a) \neq \perp$, $\exists \omega \in \Sigma_d^*$ s.t. $q'_d = \hat{\delta}_d(q_d, a\omega)$, $q'_d R q'_f \wedge \lambda_f(q_f, a) = \hat{\lambda}_d(q_d, a\omega)$,

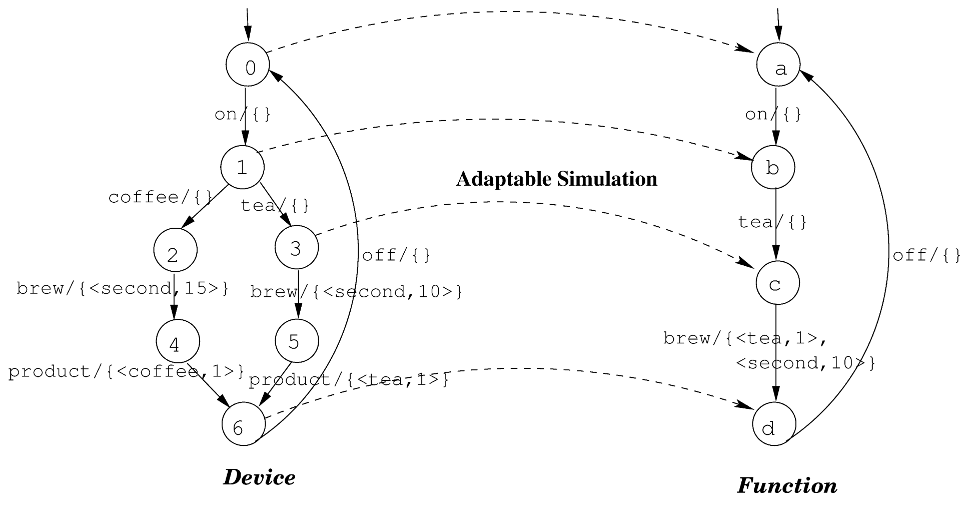


Fig. 5. An adaptive simulation relation.

where the signal a is called *matched* signal, the sequence of signals ω are called *automated* signals, and the rest of signals in Σ_d other than matched and automated ones are called *disabled* signals.

In the definition of specification matching, behavioral matching is defined through $q'_d R q'_f$, where $q'_d = \hat{\delta}_d(q_d, a\omega)$, $q'_f = \delta_f(q_f, a) \neq \perp$, and $q_d R q_f$; and functional matching is defined through $\lambda_f(q_f, a) = \hat{\lambda}_d(q_d, a\omega)$. Based on the `S-matching` relation, the input signals of a given device can be differentiated into three sets: matched signals, automated signals, and disabled signals.

Fig. 5 illustrates an example of S-matching relation:

$$R = \{(0,a), (1,b), (3,c), (6,d)\}.$$

Consider the transition

$$c \xrightarrow{brew/\{\langle tea,1\rangle, \langle second,10\rangle\}} d$$

in $\mathcal{F}$ and the corresponding transitions

$$3 \xrightarrow{brew/\{\langle second,10\rangle\}} 5 \xrightarrow{product/\{\langle tea,1\rangle\}} 6$$

in $\mathcal{D}$. Both of them have the same starting input label `brew` and the same functional behaviors (output labels) `{<tea,1>,<second,10>}`, which is consistent to the (2) of the S-matching definition.

3.3. Adaptability

The main problem of adaptability is to decide whether there exists an adapter, as informally addressed in Section 2.1, which is able to adapt a given device to match all the specifications of a given design function. We formalize the definitions of an adapter, a valid adapter, and adaptability as follows.

Definition 3 *(Adapter).* An *adapter* is described by a five-tuple labeled transition system $\langle Q, \Sigma_e, \Sigma_i, q_0, \delta\rangle$, where

(1) Q is a finite set of states, $q_0 \in Q$ is a unique start state;
(2) Σ_e is a finite set of external actions—those actions which can be observed by the users;
(3) Σ_i is a finite set of internal actions—those actions which cannot be observed by the users;
(4) $\delta : Q \times (\Sigma_e \cup \Sigma_i) \to Q_\perp$ is a transition partial function that takes as arguments a state and an input signal (either external or internal), returning a state or $\perp$ meaning that there is no such transition.

The external actions in Σ_e correspond to matched signals in device adaptation, while the internal actions in Σ_i represent automated signals. In the diagram representation of an adapter, we usually display internal actions paired with [] to distinguish them from external actions. Additionally, the function δ can also be extended to a function $\hat{\delta}$ mapping $Q \times (\Sigma_e \cup \Sigma_i)^*$ to $Q_\perp$ defined similar to the function δ in ALTS.

Definition 4 *(Valid adapter).* Given an ALTS of device $\mathcal{D} = \langle Q_d, \Sigma_d, \delta_d, q_{d_0}, \Gamma_d, \lambda_d\rangle$, a *valid adapter* $\mathcal{I} = \langle Q, \Sigma_e, \Sigma_i, q_0, \delta\rangle$ of $\mathcal{D}$ is an adapter with the following constraints:

(1) $\Sigma_e \cup \Sigma_i \subseteq \Sigma_d$;
(2) there is a mapping function $M_d : Q \to Q_d$ such that for any state $p \in Q$, if $\delta(p, a) \neq \perp$ for some $a \in \Sigma_e \cup \Sigma_i$, then $\delta_d(M_d(p), a) = M_d(\delta(p, a))$;
(3) for any state $q \in Q$, if there exists a signal $a \in \Sigma_i$ such that $\delta(q, a) \neq \perp$, then for any other signal $a_1 \in \Sigma_i \cup \Sigma_e$ satisfying $a_1 \neq a$, we have $\delta(q, a_1) = \perp$.

Definition 4 specifies the constraints for an adapter which can be used for adapting a given device. Constraint (2) shows that for any two states $p \in Q$ and $q \in Q_d$ such that $q = M_d(p)$, if p has a transition labeled a to a next state $\delta(p, a)$, then q must have the same labeled transition to a next state $\delta_d(q, a)$, and their next states are also mapped through the function M_d, that is, $M_d(\delta(p, a)) = \delta_d(q, a)$. Constraint (3) guarantees that with a valid adapter, it is impossible to have both internal (automated) actions and external (matched) actions out of any adapter states. Otherwise, the adapted device will have uncertainty at those states. For the same reason, at most one internal action is allowed from any adapter state.

Definition 5 *(Adaptability).* Given ALTSs $\mathcal{F} = \langle Q_f, \Sigma_f, \delta_f, q_{f_0}, \Gamma_f, \lambda_f\rangle$ and $\mathcal{D} = \langle Q_d, \Sigma_d, \delta_d, q_{d_0}, \Gamma_d, \lambda_d\rangle$. We say that $\mathcal{D}$ is *adaptable* to implement $\mathcal{F}$ if there exists a valid adapter $\mathcal{I} = \langle Q, \Sigma_e, \Sigma_i, q_0, \delta\rangle$ with a mapping function $M_d : Q \to Q_d$ for the device $\mathcal{D}$ such that:

(1) $\Sigma_e = \Sigma_f$;
(2) there is an one-one mapping function $M_f : Q_f \to Q$, such that given any two states $q_f, q'_f \in Q_f$, if $\delta_f(q_f, a) = q'_f$ for some $a \in \Sigma_f$, then there exists a sequence of signals $\omega \in \Sigma_i^*$ such that:
   (a) $\hat{\delta}(M_f(q_f), a\omega) = M_f(q'_f)$, and
   (b) $\lambda_f(q_f, a) = \hat{\lambda}_d(M_d(M_f(q_f)), a\omega)$.

Definition 5 gives a formal reason why a device $\mathcal{D}$ can be adapted to a design function $\mathcal{F}$ through a valid adapter $\mathcal{I}$ of $\mathcal{D}$. The reason is based on the existence of two mapping functions M_d and M_f. The mapping function M_d ensures that the adapter $\mathcal{I}$ is a valid one for the device $\mathcal{D}$. And the mapping function M_f makes sure both behavioral and functional matching between $\mathcal{D}$ and $\mathcal{F}$. That is, for each transition with an input signal a and attributes o in $\mathcal{F}$, the corresponding transitions in $\mathcal{I}$ and $\mathcal{D}$ start with a transition with a matched signal a (see (2)(a) in Definition 5), and their accumulated attributes for the corresponding transitions in $\mathcal{D}$ is same as o (see (2)(b) in Definition 5).

Fig. 6 illustrates an example where a device $\mathcal{D}$ is adaptable to implement a design function $\mathcal{F}$ through an adapter $\mathcal{I}$. Each transition in $\mathcal{F}$ can be implemented by its corresponding transitions in $\mathcal{D}$ with a two-step mapping procedure M_f and M_d. For instance, the transition

$$c \xrightarrow{brew/\{\langle tea,1\rangle,\langle second,10\rangle\}} d$$

in $\mathcal{F}$ is implemented using

$$3 \xrightarrow{brew/\{\langle second,10\rangle\}} 5 \xrightarrow{product/\{\langle tea,1\rangle\}} 6$$

in $\mathcal{D}$ with the same functionalities (output labels); and the same behaviors (input labels) are achieved through the adapter by starting with the same input and then automating the internal action `product`. State $(5, c)_0$ in $\mathcal{I}$ represents an internal state, i.e., an state that is not observable by an external observer.

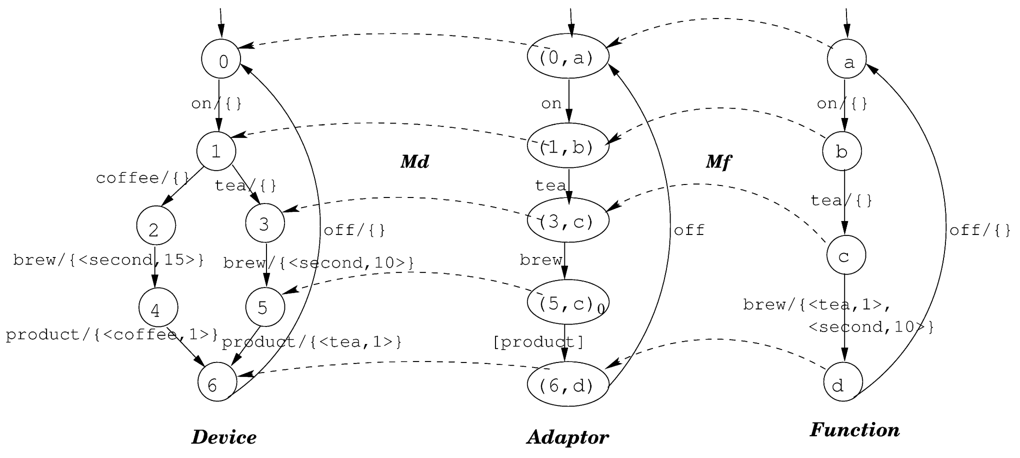


Fig. 6. An example of adaptability.

3.4. Component adaptation

This subsection explains how to achieve adaptation once the adapter is constructed. In our specification matching framework, component adaptation is achieved through the interaction between the adapter $\mathcal{I}$ and the device $\mathcal{D}$. This interaction is formally defined by a new parallel composition operator $//$, which combines a forced action in $\mathcal{I}$ with a corresponding transition in $\mathcal{D}$ that is forced to give an unobservable τ action in $\mathcal{I}//\mathcal{D}$. In addition, the $//$ operator combines an external action in $\mathcal{I}$ with an identical external action in $\mathcal{D}$ resulting in an observable external move in $\mathcal{I}//\mathcal{D}$. The $//$ operator is defined as follows:

Definition 6 *(Adaptation).* Given a valid adapter $\mathcal{I} = \langle Q, \Sigma_e, \Sigma_i, q_{i_0}, \delta\rangle$ and a device $\mathcal{D} = \langle Q_d, \Sigma_d, \delta_d, q_{d_0}, \Gamma_d, \lambda_d\rangle$, $\mathcal{I}//\mathcal{D}$ is defined to be a process described by the ALTS $\langle Q_{I//D}, \Sigma_{I//D}, \delta_{I//D}, q_0, \Gamma_{I//D}, \lambda_{I//D}\rangle$, where

- $Q_{I//D} = Q_d \times Q$;
- $\Sigma_{I//D} = \Sigma_e \cup \{\tau\}$;
- $q_0 = (q_{d_0}, q_{i_0})$;
- $\Gamma_{I//D} = \Gamma_d$;
- $\delta_{I//D}$ and $\lambda_{I//D}$ are defined by the following rules:

Automated move: $\mathcal{I}//\mathcal{D}$ makes an unobservable τ move, when $\mathcal{I}$ forces a transition in $\mathcal{D}$:

$$\frac{q_d \xrightarrow{a/\lambda(q_d,a)} q_{d1},\ q_i \xrightarrow{[a]} q_{i1}}{(q_d, q_i) \xrightarrow{\tau/\lambda(q_d,a)} (q_{d1}, q_{i1})}$$

Matched move: $\mathcal{I}//\mathcal{D}$ makes an observable move, with both $\mathcal{D}$ and $\mathcal{I}$ simultaneously responding to the same external signal:

$$\frac{q_d \xrightarrow{a/\lambda(q_d,a)} q_{d1},\ q_i \xrightarrow{a} q_{i1}}{(q_d, q_i) \xrightarrow{a/\lambda(q_d,a)} (q_{d1}, q_{i1})}$$

Otherwise: For any other combination of $q \in Q_{I//D}$ and $a \in \Sigma_{I//D}$, we have $\delta(q,a) = \bot$ and $\lambda(q,a) = \bot$ as well.

This composition operator $//$ is different from the one defined in [29] in that the new operator differentiates the functionalities from the behaviors on each transition. Only the behaviors are shadowed on the forced moves, while the functionalities are properly inherited to the composition. Additionally, the $//$ operator is quite different from synchronous parallel || operator of CCS [21]. First, unlike synchronous parallel operator, the new operator disallows autonomous moves. Secondly, forced move is different from synchronous parallel with global hiding, as forcing essentially leads to *state-based hiding*.

For $\mathcal{D}$ to match $\mathcal{F}$, $\mathcal{D}$ and $\mathcal{F}$ have to satisfy the adaptability (Definition 5); on the other hand, $\mathcal{D}//\mathcal{I}$ must be equivalent to $\mathcal{F}$ in both behavioral and functional aspects. Behavior equivalence is checked by using Milner's weak bisimulation [21] (denoted as $\approx$ and also known as observational equivalence); functional equivalence is checked by the following ϕ-rule: Given $q \approx q_f$ (behavioral equivalent) for any $q \in Q_{D//I}$ and $q_f \in Q_f$, if $\delta_f(q_f, a) = q'_f$ for some $a \in \Sigma_f$, then there exists a sequence of signals $\omega \in \Sigma^*_{D//I}$ satisfying:

(1) $\hat{\delta}(q, a\omega) = q'$ such that $q' \approx q'_f$; and
(2) $\lambda_f(q_f, a) = \hat{\lambda}_{D//I}(q, a\omega)$.

Thus, an alternative definition for Adaptability (Definition 5) is given as follows based on the composition operator $//$:

Definition 7. A device D can implement a function F (D matches F) if there exists an adapter I such that $(I//D)$ and F are equivalent in both behavioral and functional aspects, where behavioral equivalence is checked by using Milner's weak bisimulation, and functional equivalence is checked by using ϕ-rule defined as above.

Intuitively, $\mathcal{D}$ is adaptable to implement $\mathcal{F}$ if the behaviors of $\mathcal{D}//\mathcal{I}$ and $\mathcal{F}$ cannot be distinguished by an external observer.

3.5. Adaptability checking via specification matching

For convenience, we assume that $\mathcal{I} = \langle Q, \Sigma_e, \Sigma_i, q_0, \delta\rangle$ is a valid adapter for the device $\mathcal{D} = \langle Q_d, \Sigma_d, \delta_d, q_{d_0}, \Gamma_d, \lambda_d\rangle$ to implement the design function $\mathcal{F} = \langle Q_f, \Sigma_f, \delta_f, q_{f_0}, \Gamma_f, \lambda_f\rangle$, and $M_d : Q \to Q_d$ and $M_f : Q_f \to Q$ are its two associated mapping functions. Thus, we have the following properties.

Lemma 1. *For any state $p \in Q$ in $\mathcal{I}$, if $\hat{\delta}(p, \omega) \neq \perp$ for some $\omega \in (\Sigma_e \cup \Sigma_i)^*$, then $\hat{\delta}_d(M_d(p), \omega) = M_d(\hat{\delta}(p, \omega))$.*

Proof. This lemma can be easily proved based on Definition 4(2) by the induction on the length of ω. □

Lemma 2. *Given two states $q_f, q'_f \in Q_f$ such that $\delta_f(q_f, a) = q'_f$ for some $a \in \Sigma_f$, there exists a sequence of signals $\omega \in \Sigma^*_d$ such that*

$$\hat{\delta}_d\big(M_d\big(M_f(q_f)\big), a\omega\big) = M_d\big(M_f(q'_f)\big).$$

Proof. According to Definition 5, there exists a sequence of signals $\omega \in \Sigma^*_i \subseteq \Sigma^*_d$ satisfying

$$\begin{aligned}
&\hat{\delta}\big(M_f(q_f), a\omega\big) = M_f(q'_f) \\
&\quad \Rightarrow M_d\big(\hat{\delta}\big(M_f(q_f), a\omega\big)\big) = M_d\big(M_f(q'_f)\big) \quad \text{applying } M_d \text{ on both sides} \\
&\quad \Rightarrow \hat{\delta}_d\big(M_d\big(M_f(q_f)\big), a\omega\big) = M_d\big(M_f(q'_f)\big) \quad \text{Lemma 1.} \qquad \square
\end{aligned}$$

Lemma 2 extends the behavioral matching property from between $\mathcal{F}$ and $\mathcal{I}$ to between $\mathcal{F}$ and $\mathcal{D}$ through the composition of M_d and M_f.

Theorem 1. *If $\mathcal{D}$ is adaptable to implement $\mathcal{F}$ with a valid adapter $\mathcal{I}$ and its two associated mapping functions M_d and M_f, then $R = \{(M_d(M_f(q_f)), q_f) \mid q_f \in Q_f\}$ is an S-matching relation.*

Proof. We show how R satisfies Definition 2.

(i) $\forall q_f \in Q_f$, there exists $M_d(M_f(q_f)) \in Q_d$ such that $(M_d(M_f(q_f)), q_f) \in R$;

(ii) for any $(q_d, q_f) \in Q_d \times Q_f$, $q_d R q_f$ implies $q_d = M_d(M_f(q_f))$ according to the definition of R. Then, for $\forall a \in \Sigma_f$ s.t. $q'_f = \delta_f(q_f, a) \neq \bot$, according to Definition 5 (Adaptability), there exists a sequence of signals $\omega \in \Sigma_i^* \subseteq \Sigma_d^*$ such that:

(1) $\lambda_f(q_f, a) = \hat{\lambda}_d(M_d(M_f(q_f)), a\omega) = \hat{\lambda}_d(q_d, a\omega)$ (functional matching);

(2)

$$\begin{aligned} M_d(M_f(q'_f)) &= \hat{\delta}_d(M_d(M_f(q_f)), a\omega) \quad \text{Lemma 2} \\ &= \hat{\delta}_d(q_d, a\omega) \quad \text{since } q_d = M_d(M_f(q_f)) \\ &= q'_d \quad \text{let } q'_d = \hat{\delta}_d(q_d, a\omega) \quad \text{(behavioral matching).} \end{aligned}$$

Thus, we have $q'_d R q'_f$ based on the definition of R.

Therefore, R satisfies Definition 2, which completes the proof. □

Theorem 1 shows that the existence of an S-matching relation is a necessary condition for the existence of adaptability from $\mathcal{D}$ to $\mathcal{F}$ via a valid adapter $\mathcal{I}$. Both functional matching and behavioral matching between $\mathcal{D}$ and $\mathcal{F}$ can be derived by applying the mapping functions M_d and M_f.

Theorem 2. *If there exists an S-matching relation R from $\mathcal{D}$ to $\mathcal{F}$, then there exists a valid adapter $\mathcal{I}$ such that $\mathcal{D}$ is adaptable to implement $\mathcal{F}$.*

Proof. The adapter $\mathcal{I} = \langle Q, \Sigma_e, \Sigma_i, q_0, \delta\rangle$ with its two associated mapping functions M_d and M_f can be constructed as follows:

(1) $Q = Q_e \cup Q_i$, where $Q_e = \{(q_d, q_f) \mid q_d R q_f\}$ and Q_i is shown in the next item;
(2) Σ_i, Q_i and δ can be constructed in the following way: $\forall q_f, q'_f \in Q_f$ s.t. $\delta_f(q_f, a) = q'_f$ for some matched signal $a \in \Sigma_f$, let $q_d R q_f$ and $q'_d R q'_f$ (that is, $(q_d, q_f) \in Q_e$ and $(q'_d, q'_f) \in Q_e$), according to Definition 2, there exists an automated signal sequence $\omega \in \Sigma_d^*$ such that $q'_d = \hat{\delta}_d(q_d, a\omega)$,
- If $\omega = \epsilon$, $\delta((q_d, q_f), a) = (q'_d, q'_f)$;
- If $\omega \neq \epsilon$, let $\omega = a_1 a_2 \cdots a_n$, where $n > 0$ and $a_j \in \Sigma_i$ for any $1 \leqslant j \leqslant n$, we have $(\hat{\delta}_d(q_d, a\omega_k), q'_f)_k \in Q_i$ where $\omega_k = a_1 \cdots a_k$ and $0 \leqslant k < n$ ($\omega_0 = \epsilon$), and δ for $0 \leqslant m < n$:

$$\begin{aligned} &\delta((q_d, q_f), a) = (\delta_d(q_d, a), q'_f)_0 = (\hat{\delta}_d(q_d, a\omega_0), q'_f)_0, \\ &\delta((\hat{\delta}_d(q_d, a\omega_m), q'_f)_m, a_{m+1}) = (\hat{\delta}_d(q_d, a\omega_{m+1}), q'_f)_{m+1}, \\ &\delta((\hat{\delta}_d(q_d, a\omega_{n-1}), q'_f)_{n-1}, a_n) = (q'_d, q'_f). \end{aligned}$$

For any other combination of $q \in Q$ and $a \in \Sigma_i \cup \Sigma_e$, we have $\delta(q, a) = \bot$; The purpose that each state in Q_i uses a subscript is to make itself a distinct state name.

(3) $q_0 = (q_{d_0}, q_{f_0})$, where $q_{d_0} R q_{f_0}$; (For simplicity, we assume that $q_{d_0} R q_{f_0}$. Otherwise, a sequence of automated signals might be used to find an new initial state having the relation R with f_0.)
(4) M_d is defined as the following:
- $M_d((q_{d_0}, q_{f_0})) = q_{d_0}$;
- $\forall p, q \in Q$ s.t. $\delta(p, a) = q$ and $M_d(p) = p_d$ for some signal $a \in \Sigma_e \cup \Sigma_i$, we have $M_d(q) = \delta_d(p_d, a)$.

(5) $M_f(q_f) = (q_d, q_f)$ for any $q_f \in Q_f$, where $q_d R q_f$;
(6) $\Sigma_e = \Sigma_f$.

We then prove that $\mathcal{I}$ is a valid adapter by showing how the three constraints in Definition 4 are satisfied:

- $\Sigma_i \subseteq \Sigma_d$ since in the item (2), every single $a_j \in \Sigma_i$ is originally from the automated signal sequence $\omega \in \Sigma_d^*$. $\Sigma_e \subseteq \Sigma_d$ since $\Sigma_e = \Sigma_f$, and $\Sigma_f \subseteq \Sigma_d$ due to the behavioral matching given in the S-matching relation from $\mathcal{D}$ to $\mathcal{F}$.
- The construction of M_d in the item (4) ensures the satisfaction of the second constraint in Definition 4. The proof is trivial.
- As δ is defined in the item (2), there is only one defined transition from each internal state in Q_i, and every internal state in Q_i is generated with a subscript to make itself a distinct state name. Therefore, the third constraint is also satisfied.

Finally, we conclude that $\mathcal{D}$ is adaptable to implement $\mathcal{F}$, according to Definition 5 (Adaptability), because there exists a valid adapter $\mathcal{I}$ with two mapping functions M_d and M_f such that:

- $\Sigma_e = \Sigma_f$, as given in the item (6); and
- Consider two states $q_f, q'_f \in Q_f$ if $\delta_f(q_f, a) = q'_f$ for some $a \in \Sigma_f$. Let $q_d R q_f$ and $q'_d R q'_f$, where $q_d, q'_d \in Q_d$. Then, there exists a sequence of signals $\omega \in \Sigma_i^*$ satisfying:

$$\begin{aligned}\hat{\delta}(M_f(q_f), a\omega) &= (q'_d, q'_f) \quad \text{according to item (2)}\\ &= M_f(q'_f) \quad \text{according to item (5)}\end{aligned}$$

and

$$\begin{aligned}\lambda_f(q_f, a) &= \hat{\lambda}_d(q_d, a\omega) \quad \text{according to S-matching relation } R\\ &= \hat{\lambda}_d\big(M_d\big((q_d, q_f)\big), a\omega\big) \quad \text{according to item (4)}\\ &= \hat{\lambda}_d\big(M_d\big(M_f(q_f)\big), a\omega\big). \quad \text{according to item (5).} \qquad \square\end{aligned}$$

Theorem 2 shows that the existence of an S-matching relation is also a sufficient condition for the existence of adaptability from $\mathcal{D}$ to $\mathcal{F}$. As shown in the proof, a valid adapter $\mathcal{I}$ and two associated mapping functions M_d and M_f can be constructed based on the given S-matching relation R. And the adapter $\mathcal{I}$ is constructed in such a way that matched signals in the S-matching relation have to be defined as external actions to simulate a given design function, automated signals are masked as internal actions, and disabled signals are simply ignored since they will not be used after adaptation.

Theorem 3 *(Adaptability using specification matching). Given a device $\mathcal{D}$ and a function $\mathcal{F}$, $\mathcal{D}$ is adaptable to implement $\mathcal{F}$ if and only if there exists an S-matching relation between $\mathcal{D}$ and $\mathcal{F}$.*

Proof. Based on Theorems 1 and 2. □

Theorem 3 gives the theoretical foundation for applying specification matching on adaptive reuse in the domains of embedded systems. To see whether there exists an adaptive reuse from a device $\mathcal{D}$ to a design function $\mathcal{F}$, we can simply check the S-matching relation between $\mathcal{D}$ and $\mathcal{F}$ instead of constructing the adapter directly. If the S-matching relation does exist, we can then construct the adapter automatically based on the S-matching relation.

4. Logic-based S-matching tool

Theorem 3 gives a theoretical foundation for our logic-based specification matching tool (S-matching tool in short). Component adaptability can be identified by checking a pure mathematical S-matching relation between components, and further an adapter can be constructed based on the S-matching relation. Given two ALTSs: a device $\mathcal{D}$ and a design function $\mathcal{F}$, the S-matching tool will generate an automated adapter $\mathcal{I}$ if $\mathcal{D}$ can be adapted to implement $\mathcal{F}$, otherwise, a partial adapter with error feedback will be shown.

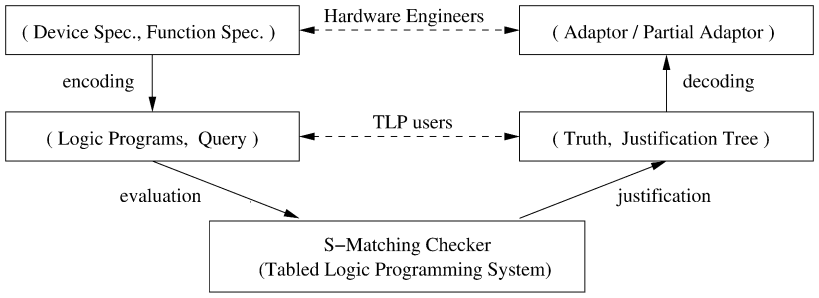

(Device Spec., Function Spec.)
Hardware Engineers
(Adaptor / Partial Adaptor)
encoding
decoding
(Logic Programs, Query)
TLP users
(Truth, Justification Tree)
evaluation
justification
S–Matching Checker
(Tabled Logic Programming System)

Fig. 7. An architecture for S-matching tool.

4.1. Architecture

The S-matching tool can be described as the architecture shown in Fig. 7, which is an overview of the interaction between specification matching and a tabled logic programming (TLP) system. The S-matching tool, from the given specification of a device and a design function to the adapter generation, is reduced elegantly to S-matching checker with tabled logic programming. The S-matching checker, running on a TLP system, establishes the truth value of the goal by query evaluation, and then justifies the truth value to provide evidence, in terms of a proof. The reduction includes a forward step, which encodes the specification of both device and function as logic programs, and a backward step, which decodes the query result and justification providing the adapter or a partial adapter if S-matching fails.

We adopt XSB [35] system to implement the S-matching tool. XSB is a logic programming system that extends Prolog-style resolution with *tabled resolution* [6,32]. Traditional logic programming systems (e.g., Prolog) use SLD resolution [15] with the following *computation strategy*: subgoals of a resolvent are solved from left to right and clauses that match a subgoal are applied in the textual order they appear in the program. It is well known that SLD resolution may lead to non-termination for certain programs, even though an answer may exist via the declarative semantics. That is, given any static computation strategy, one can always produce a program in which no answers can be found due to non-termination even though some answers may logically follow from the program. In case of Prolog, programs containing certain types of left-recursive clauses are examples of such programs.

The XSB system eliminates such infinite loops by extending logic programming with tabled resolution. The main idea of tabled resolution is to memoize the answers to some calls and use the memoized answers to resolve subsequent variant calls.[4] Tabled resolution adopts a dynamic computation strategy while resolving subgoals in the current resolvent against matched program clauses or tabled answers. It keeps track of the nature and type of the subgoals; if the subgoal in the current resolvent is a variant of a former tabled call, tabled answers are used to resolve the subgoal; otherwise, program clauses are used following SLD resolution.

The main advantages of tabled resolution are that a TLP system terminates more often by computing fixed-points, avoids redundant computation by memoing the computed answers, and keeps the declarative and procedural semantics consistent for pure logic programs with bounded-size terms. A tabled logic programming system can be thought of as an engine for efficiently computing fixed-points, which is critical for many practical applications. XSB has already been used for building efficient model checkers [26] and bisimulation checkers [4]. More importantly, the XSB system provides a *justifier* [23], which essentially constructs concise evidence or debugging information to support the results of query evaluation. This justifier can be easily extended to provide a solution to generate automated adapter for reusing hardware/software components.

In a tabled logic programming system, only tabled predicates are resolved using tabled resolution. Tabled predicates are explicitly declared as follows:

```
:- table p/n.
```

where `p` is a predicate name and `n` is its arity. A global data structure *table* [6,32] is introduced to memoize the answers of any subgoals to tabled predicates, and to avoid their recomputation.

[4] We say two Prolog calls C_1 and C_2 are variant if there exist substitutions ϕ and σ such that $C_1 = C_2\phi$ and $C_2 = C_1\sigma$.

Consider a simple tabled Prolog program checking reachability as follows.

```
:- table reach/2.
reach(X, X).
reach(X, Y) :- arc(X, Z), reach(Z, Y).
reach(X, Y) :- arc(X, Y).
arc(a, b). arc(b, a). arc(b, c).
:- reach(a, X).
```

Without tabling `:- table reach/2`, this program cannot terminate in a traditional Prolog system due to the left-recursive definition of `reach/2` and cyclic arcs between `a` and `b`. To solve such a non-termination program in a traditional Prolog system, the programmer has to code the traversal of the graph (arcs) explicitly to somehow maintain visited nodes. However, using a tabled Prolog system, the programmer does not need to worry about such a termination problem. At the code level, it is *devoid* of all graph traversal. All the programmer needs to do is to define the reachability relation correctly, how the traversal of graph is solved is hidden at the system level. For the above example, the tabled Prolog gives the correct answers: `X = a`, `X = b` and `X = c`, and then terminates.

4.2. Encoding ALTS

An ALTS $A = \langle Q, \Sigma, \delta, q_0, \Gamma, \lambda\rangle$ is encoded as a set of facts in a logic program such that whenever $\delta(p, a) = q$ and $\lambda(p, a) = \{\langle n_1, v_1\rangle, \ldots, \langle n_k, v_k\rangle\}$, where $k \geqslant 0$, $\mathtt{trans}(p, a, [(n_1, v_1), \ldots, (n_k, v_k)], q)$ is a fact. Predicate `trans/4` encodes a given transition where the first and the last parameters indicate the source and destination states, the second parameter a indicates the input signal that triggers this transition, and the third one is the attributes associated with the transition. An example coding of the coffee/cappuccino brewer ALTS (see Fig. 4(a)) is shown below as a sequence of facts:

```
trans(0, on, [], 1).
trans(1, coffee, [(coffee, 1)], 2).
trans(1, cappuccino, [(cappuccino, 1)], 3).
trans(2, produce, [(second, 30)], 4).
trans(3, produce, [(second, 15)], 4).
trans(4, off, [], 0).
```

Notice that in S-matching tool, we use predicate `transD/4` and `transF/4` to represent the ALTSs device $\mathcal{D}$ and $\mathcal{F}$ respectively. The predicates `transD/4` and `transF/4` are defined similar to the definition of the predicate `trans/4`. And we assume that all states in the ALTS are connected.

The number of facts in a coding of ALTS is proportional to the number of transitions in the ALTS. However, the number of transitions is generally quadratic to the number of states since in most practical ALTSs or LTSs for embedded systems, there is at most one transition from one state to another, that is, a control state with a different labeled transition usually reaches a different control state. Otherwise, there is no deterministic relation between the number of transitions and the number of states.

4.3. S-matching checker

Rather than encoding the S-matching relation directly, it is more convenient to encode the dual of S-matching relation as a tabled logic program within XSB. The reason is that XSB is a least fixed point engine whereas S-matching computation is based on a greatest fixed point (the greatest S-matching relation between $\mathcal{F}$ and $\mathcal{D}$ is computed when it exists). The rules for a pair of states to be non-S-matching are directly encoded as XSB clauses.

In the following, we provide the formal definition and the corresponding encoding in XSB. Let R be an S-matching relation, then $\overline{R}$, the dual relation of R, is defined as follows.

```
:- table nsmatch/2.
:- table attr/4.
:- table findD/3.

nsmatch(D, F) :-
 transF(F, I, A, F1),
 findall(D1,
   attr(D, I, D1, A), L),
 all_nsmatch(L, F1).

all_nsmatch([], _).
all_nsmatch([D|L], F) :-
 nsmatch(D, F),
 all_nsmatch(L, F).

attr(D, I, D1, A) :-
 transD(D, I, A1, D2),
 sub(A, A1, AR),
 findD(D2, D1, AR).
```

```
findD(D, D, []).
findD(D, D2, A) :-
 transD(D, _, A1, D1),
 sub(A, A1, AR),
 findD(D1, D2, AR).

sub(A, [], A).
sub(A, [Pair|L], A1) :-
 sub1(A, Pair, A2),
 sub(A2, L, A1).

sub1([(N,V)|L], (N,V), L).
sub1([(N,V1)|L], (N,V2),
        [(N,V3)|L]) :-
 V3 is V1-V2,
 V3 > 0, !.
sub1([A|L], (N,V), [A|L1])
 :- sub1(L, (N,V),L1).
```

Fig. 8. Coding for S-matching checker.

Definition 8 *(Dual of S-matching).* Given two ALTSs: $\mathcal{F} = \langle Q_f, \Sigma_f, \delta_f, q_{f_0}, \Gamma_f, \lambda_f\rangle$, $\mathcal{D} = \langle Q_d, \Sigma_d, \delta_d, q_{d_0}, \Gamma_d, \lambda_d\rangle$, $\forall(q_d, q_f) \in Q_d \times Q_f$, $q_d\overline{R}q_f$ if

- $\exists a \in \Sigma_f$ s.t. $q'_f = \delta_f(q_f, a) \neq \bot$,
  $\forall\omega \in \Sigma_d^*$ s.t. $q'_d = \hat{\delta}_d(q_d, a\omega) \wedge \lambda_f(q_f, a) = \hat{\lambda}_d(q_d, a\omega)$, $q'_d\overline{R}q'_f$.

This definition is equivalent to Definition 2 but emphasizes the fact that a pair (q_d, q_f) is *not S-matching* if for a transition a from q_f in $\mathcal{F}$, all reachable states q'_d from q_d in $\mathcal{D}$ with some transition sequence $a\omega$ cannot simultaneously satisfy the following conditions: (i) $a\omega$ has the same output attributes as that with a in $\mathcal{F}$, and (ii) $q'_dRq'_f$.

To implement the S-matching checker, we use the predicate name `nsmatch/2` as the dual relation $\overline{R}$. The least model computation of $\overline{R}$ can be encoded as shown in Fig. 8, where `nsmatch/2`, `attr/4`, and `findD/4` are tabled predicates. Predicate `attr(D,I,D1,A)` is to find a reachable state `D1` from the state `D` through a sequence of transitions, whose first input label is `I` and whose total attributes equal to `A`. Predicate `findall(X, Goal, List)` collects all instances of `X` to `List` such that `Goal` is provable. If `Goal` is not provable, `List` will be an empty list `[]`. Predicate `all_nsmatch(L, Q)` is used to verify the relation $\overline{R}$ between each state in the given list `L` with the state `Q` in $\mathcal{F}$. `sub/3` and `sub1/3` are auxiliary predicates to verify the attributes equivalence. And `!` is the cut operation for efficiency purposes.

The generated code was very compact and extremely readable since it is devoid of all graph traversal and computation intensive routines, as these are automatically performed by the tabled logic programming engine.

Our S-matching tool takes longer to succeed than to fail in general. Since we implement the dual of S-matching, the failure case represents a successful match. The main reason why failure takes less time is as follows: consider the code of `nsmatch/2` shown in Fig. 8. To make nsmatch succeed, `all_nsmatch(L, F1)` has to be true, which means that `F1` has to *not* match with all the nodes in the list `L`; however, for the failure case, it is enough as long as one node in `L` matches with `F1`.

4.4. Evaluation and justification

Query evaluation of a goal with respect to a logic program establishes the truth or falsehood of the goal. Given two ALTSs $\mathcal{F}$ and $\mathcal{D}$ and the codes shown above, `nsmatch`(d_0,f_0) can be queried in XSB system to evaluate whether there is a complement of S-matching relation $d_0\overline{R}f_0$, where d_0 and f_0 are the initial states of the device $\mathcal{D}$ and the design function $\mathcal{F}$ respectively.

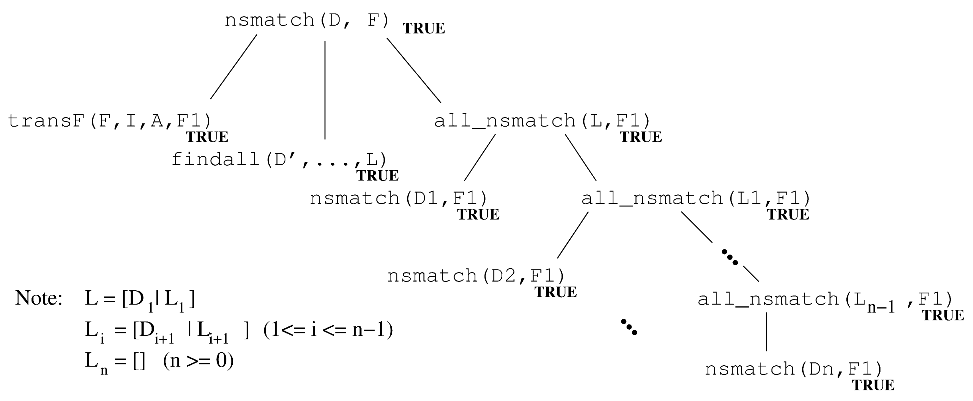


Fig. 9. Justification segment for a true literal.

However, the underlying evaluation typically provides little or no information as to why the conclusion was reached. In [23] we proposed an efficient justifier to provide evidence, in terms of proof, for the truth value of the result generated by query evaluation of a tabled logic program. The justification is based on program transformation. Specifically, every predicate in the original program corresponds to two predicates in the transformed program: an evidence predicate and a dual predicate. Whenever a query in the original program is true, the corresponding query to its evidence predicate generates the answer and its evidence simultaneously. Whenever a query in the original program fails, the corresponding query to its dual predicate generates an evidence for the failure.

Justification succinctly conveys to the user only those parts of the proof search which are relevant to the proof/disproof of the goal. The naturalness of using a tabled LP system for justification is that the answer memo tables created represent the lemmas that were tried and proved during query evaluation. By using these lemmas stored in the tables, the justifier presents only relevant parts of the derivation to the user.

The result of justification is in the form of evidence trees [23], which reflect the derivation relation between the subgoals represented by the literals. For the S-matching problem, the justification results contain sufficient information, which can be converted later to the adapter.

Fig. 9 illustrates a justification segment[5] of nsmatch(d, f), where d and f represent two state instances of $\mathcal{D}$ and $\mathcal{F}$, respectively. Each node in the segment contains a literal and its truth value, and the parent/children relation represents that the truth value of the parent literal depends on those values of its children. For example, nsmatch(d, f) is true because transF(f, i, a, $f1$), findall(d', attr(d, i, d', a), l) and all_nsmatch(l, $f1$) are true. The literals trans/3 and findall/3 are shown as leaf nodes due to the fact definitions and built-in predicates respectively, and the literal all_nsmatch(l, $f1$) is justified recursively by its child literals until all of literals are justified by true facts or built-in literals.

Similarly, Fig. 10 displays a justification segment showing the evidence why nsmatch(d, f) is false—d and f has an S-matching relation. The clause definition of nsmatch/2 implies that there are two possibilities for a nsmatch literal to fail. One is due to no transitions from state f, whose justification is shown in Fig. 10(a). That is, if a state f in the design function has no outgoing transitions, then any state in the device has an S-matching relation R with f, therefore, the $\overline{R}$ fails. The other possibility shows that for any instance transF(f,j,a,f1), we can always find d_i such that d_i is S-matched to $f1$. Notice that the node nsmatch(d, f) may contain more than three children depending on how many instances of transF(f,I,A,F1) in the given design function $\mathcal{F}$.

4.5. Decoding

The last step of the S-matching tool is to extract the adapter from the justification results. This extraction procedure is called *decoding* in the sense that contrary to encoding step, it maps the justification results in logic programming level to the proof/disproof evidence in S-matching system. If there is an S-matching relation between $\mathcal{D}$ and $\mathcal{F}$, the

[5] In the real justification trees, all the variables, such as D, F, and etc., are bound to concrete state or label instances, which are usually represented by lower-case letters such as d, f, and etc. Figs. 9 and 10 just show a skeleton of justification segments.

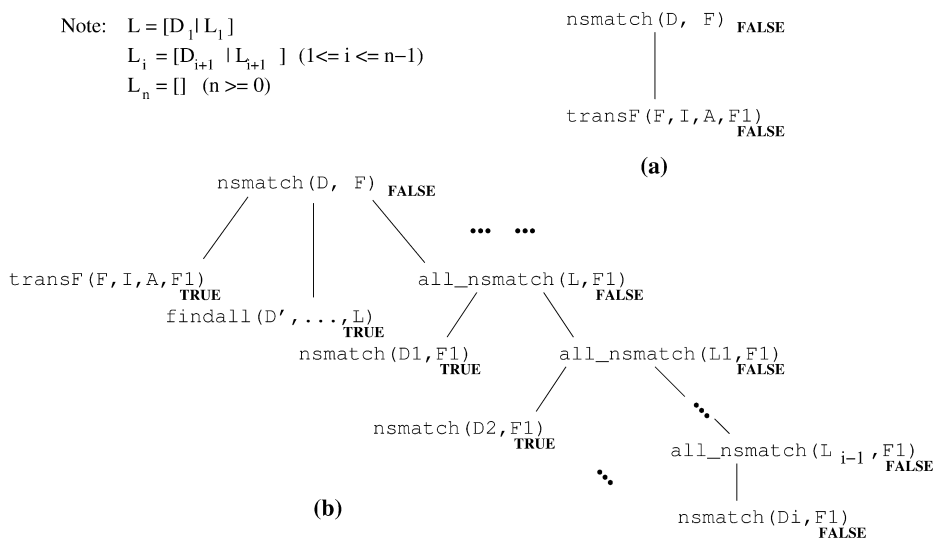


Fig. 10. Justification substructure for a false literal.

decoding step essentially generates an adapter validating the adaptation of $\mathcal{D}$ to the design function $\mathcal{F}$. Otherwise, if there is no S-matching relation, it shows a partial adapter highlighting from which states no further adaptation can be made.

The decoding step contains a few rewriting rules mapping from the possible justification segments to the primitive evidence, which, in the S-matching tool, includes matched and automated transitions. Since the implementation of S-matching checker follows the formal logic definition of $\overline{R}$ exactly, both matched and automated transitions, as defined in Definition 2(2), must be embedded in the justification results.

Let's first consider how to generate an adapter if there is an S-matching relation between given $\mathcal{D}$ and $\mathcal{F}$. Consider its two skeletons of justification segments. Fig. 10(a) shows that a state d in device can be S-matched to a state f without any outgoing transitions. Therefore, a state (d, f) without any outgoing transitions can be generated in the adapter. From Fig. 10(b), a sequence of transitions, decided by the path from d to d_i, are generated for the adapter from state (d, f) to state $(d_i, f1)$, where the label i must be the one for the first transition and the rest transitions can be easily found out following the construction method in Theorem 2.

On the other hand, if there is no S-matching relation between given $\mathcal{D}$ and $\mathcal{F}$, a partial adapter is required. The partial adapter is based on an assumption that d_0 is S-matched to f_0, where d_0 and f_0 are the start states in $\mathcal{D}$ and $\mathcal{F}$ respectively. Consider the justification of a true literal `nsmatch(d,f)` as shown in Fig. 9. This segment implies a sequence of valid transitions from state (d, f) to $(d_i, f1)$ for the partial adapter, similar to the generation of transitions for a complete adapter. Some critical failure nodes are highlighted in a partial adapter. The highlighted failure states come from the same justification segment when L is an empty list, that is, there exists a transition in $\mathcal{F}$ which has no matching transitions from $\mathcal{D}$.

Figs. 11 and 12 show a failure example of specification matching. The device $\mathcal{D}$ (Fig. 11) is a car controller having both cruise control facility as wells as acceleration through manual mode. The design function $\mathcal{F}$ (Fig. 12(a)) specifies a car controller that drives in the manual mode alone. The specification matching fails because when the controller finds belts unfastened, the device gives a two-beep alarm, while the design function needs a three-beep alarm. Notice that the attribute pairs `<cruise, 1>` and `<accl-dec, 1>` mean that the cruise control and manual control are selected once.

5. Experiments

The running time complexity of the S-matching checker is mainly determined by the computation of `nsmatch` (see Fig. 8), which is $O(|Q_f| \times |Q_d| \times m_f \times t_d \times v_f)$, where $|Q_f|$ and $|Q_d|$ denote the number of states of $\mathcal{F}$ and $\mathcal{D}$, respectively, m_f denotes the maximum number of transitions from a single node in $\mathcal{F}$, t_d denotes the total number of transitions in $\mathcal{D}$, and v_f denotes the maximum number of attribute values in $\mathcal{F}$. The first two terms $|Q_f| \times |Q_d|$

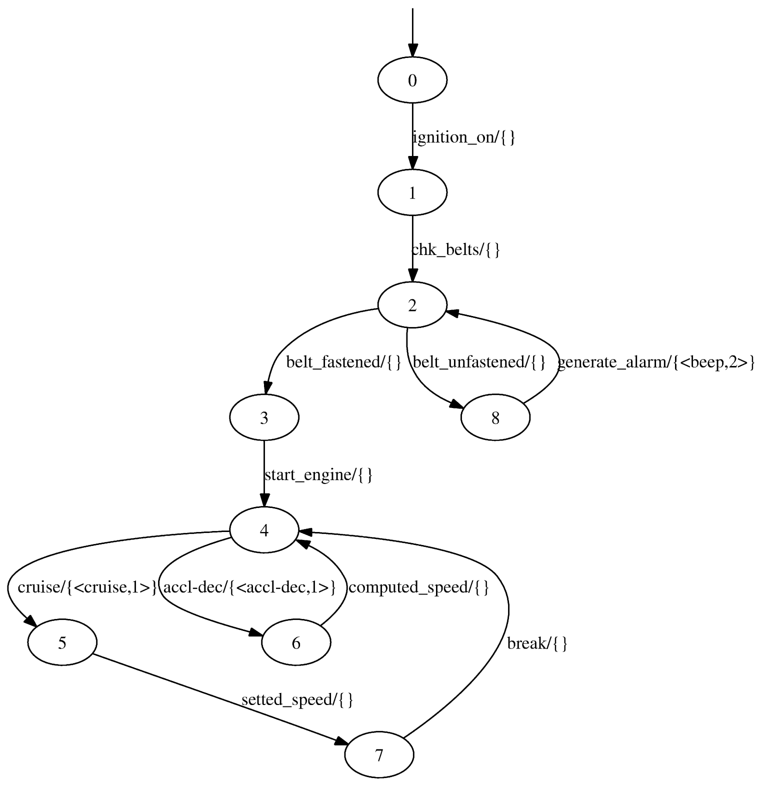


Fig. 11. Controller for a car which drives in manual and cruise modes ($\mathcal{D}$).

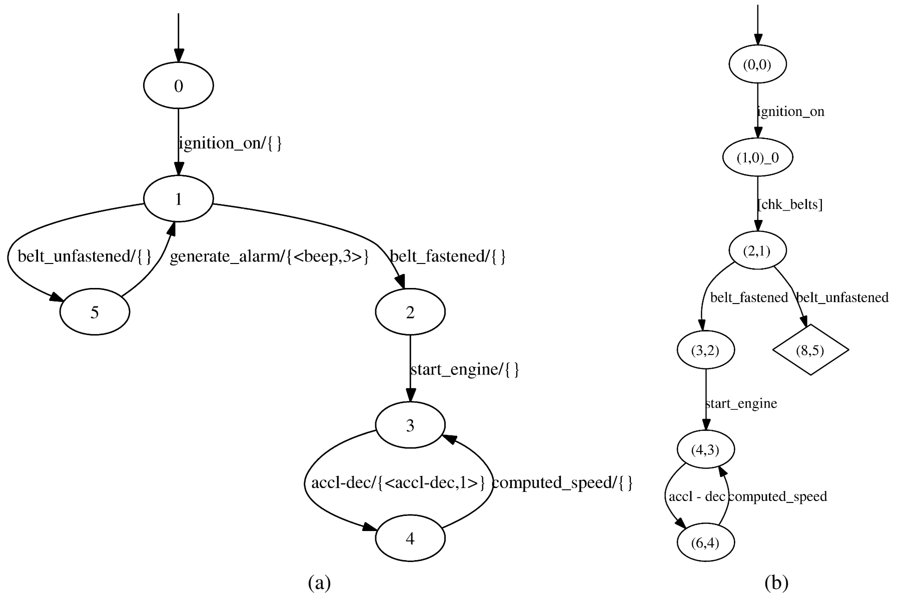


Fig. 12. (a) Simple controller for a car having manual mode alone ($\mathcal{F}$). (b) A partial adapter ($\mathcal{I}$).

arise from the recursive definition of the tabled predicate `nsmatch/2`. Since `nsmatch/2` is a tabled predicate, `nsmatch(D, F)` will be computed at most $|Q_f| \times |Q_d|$ times in the worst case for every `D` in Q_d and every `F` in Q_f; the term m_f arises from `transF(F, I, A, F1)`, which represents every transition from `F` to `F1` with an input label `I` and attributes `A`; the $t_d \times v_f$ term arises from the subgoal `findall(D1, attr(D, I, D1, A), L)` to

Table 1
Benchmarks

Benchmark	Function	Device
Test1	Simple coffee brewer States(7) Transitions(8)	Complex coffee brewer States(15) Transitions(18)
Test2	Coffee vending machine States(3) Transitions(4)	Beverage vending machine States(9) Transitions(13)
Test3	Stamp vending machine States(5) Transitions(9)	Post-accessory vending machine States(7) Transitions(15)
Test4	Car having manual mode States(4) Transitions(4)	Car having manual and cruise modes States(9) Transitions(11)
Test5	Port of a lathe controller States(13) Transitions(13)	Intel 8255 States(45) Transitions(48)
Test6	A down counter States(6) Transitions(8)	Intel 8254 States(53) Transitions(63)
Test7	Simple Image Encoder States(3) Transitions(3)	General Purpose Encoder States(13) Transitions(16)

Table 2
Experiments on successful specification matching

Benchmark	Adapter	Time (secs)
Test1	States(9) Transitions(10)	0.03
Test2	States(21) Transitions(22)	0.03
Test3	States(33) Transitions(37)	0.03
Test4	States(11) Transitions(12)	0.03
Test5	States(20) Transitions(20)	0.06
Test6	States(12) Transitions(13)	0.05
Test7	States(6) Transitions(6)	0.03

find all reachable states `D1` such that the sequence from `D` and `D1` starts with the input label `I` and has total attributes `A`, where in the worst case, the `findall` computation may go through all the transitions v_f times. In practical running instances, finding reachable states based on attr/4 is much faster because the constraints of the starting input label `I` and the bound attribute value for v_f makes the list `L` short.

Several examples, as shown in Table 1, were developed to test the S-matching tool to see how easy it was to reuse existing IPs (intellectual properties) which include both hardware and software components. Each design function or device is given with the number of states and the number of transitions.

Certain general purpose programmable devices such as Intel 8254 and Intel 8255 were selected to demonstrate component matching. These devices are normally reused by human designers by writing device drivers which supply appropriate mode and command words to select the desired mode. Automatic reuse of such programmable chips entails automation of tasks previously performed by expert human designers.

In addition to these hardware components, we tested our work on software components. A number of reactive controllers including vending machines, coffee brewers and automobile cruise controllers were also selected to illustrate the reusability of such controllers. In addition, we took a general purpose encoder developed recently [14]. Using the S-matching tool, we could automatically reuse this IP as a JPEG encoder.

All the devices and design functions were encoded using attributed labeled transition system (ALTS) specifications as $\mathcal{F}$ and $\mathcal{D}$ pairs. The control states and labeled transitions in ALTSs were derived by hand, which is actually very time consuming. Four undergraduate students, having good background in signal processing, have worked on this project for two semesters and figured out those benchmarks by reusing existing open-source IPs. Note that the number of control states are not huge. However, we are demonstrating that data transformations are standard and we are reusing the data part "as is"—the data path in $\mathcal{F}$ and $\mathcal{D}$ are identical. Our approach only changes the control part to suit the new specification.

The performance of the S-matching tool in XSB is summarized in Table 2, which represents the experiments on successful specification matching. The tool was tested on a laptop with a Pentium 4 2.4 GHz CPU and 512M RAM; the running time was measured in seconds, including both specification matching and its justification. Each generated

Table 3
Experiments on unsuccessful specification matching

Benchmark	Parital adapter	Time (secs)
Test1	States(7) Transitions(7)	0.03
Test2	States(9) Transitions(9)	0.04
Test3	States(17) Transitions(20)	0.04
Test4	States(7) Transitions(7)	0.04
Test5	States(20) Transitions(19)	0.04
Test6	States(12) Transitions(12)	0.03
Test7	States(6) Transitions(5)	0.04

adapter is given with the number of states and the number of transitions. The experimental results are mainly used to validate the correctness of our S-matching for reusing real embedded system components.

In general, our S-matching tool takes longer time to find an unsuccessful specification matching than a successful one. The experimental results on unsuccessful specification matching are shown in Table 3, where the benchmarks in each test have been changed a bit for mismatch (compared to the tests in Table 2 for successful matching). Since we implement the dual of S-matching, the failure case for a predicate query (e.g., `nsmatch`(d_0, f_0)) represents a successful specification match. Consider the code of `nsmatch/2` shown in Fig. 8. To make `nsmatch` succeed, `all_nsmatch(L, F1)` has to be true, which means `F1` has to *not* match with all the nodes in the list `L`; however, for the failure case, it is enough as long as one node in `L` matches with `F1`. However, in the practical experiments, the unsuccessful specification matching does not slow much due to the short list `L`, as explained earlier in this section.

6. Related work

In this paper, we introduced ALTS to model components both behaviorally and functionally. System behavior is captured by the ALTS transition structure that reacts to environment inputs. Functionality is captured as outputs on the transitions which are attribute-value pairs. Syntactically, ALTS looks similar to input-output boolean automata (IOBA) [18], which are used to model synchronous programs. In an IOBA, a transition triggers based on a Boolean guard over inputs and produces a set of outputs. Our transitions are simpler in the sense that they trigger based on some abstract action (similar to labelled transition systems [11]) and produce a list of output actions that are attribute pairs. The attribute pairs are syntactically like outputs generated by IOBA. However, they have special semantics that is used during the matching process to generate correct functionality. Lynch and Tuttle [16] have proposed input output automata (IOA) for the modelling of distributed systems. An IOA can capture a system or a communication channel. An IOA transition also triggers based on some environment condition and produces some outputs as a result. Unlike ALTS, which are specifically designed for embedded systems, IOA is designed for distributed systems and hence is much more general—it may be nondeterministic, can have communication using send and receive primitives and have more complex synchronization. An ALTS may be thought of as a specialization of IOAs to suit embedded systems.

Refinement is a formal approach for relating two levels of abstraction of the same design. Abadi and Lamport [1] introduced refinement mappings to check if a low-level implementation correctly refines a high-level specification using the notion of trace containment. Milner [21] proposed simulation and bisimulation over processes defined using the process algebra CCS. Simulation is a preorder that is stronger than trace containment but weaker than bisimulation equivalence (bisimulation has been shown to be the strongest notion for comparing two systems behaviorally [7]). This paper proposes S-matching relation, that is also a refinement from a specification (function $\mathcal{F}$) to an implementation (a device $\mathcal{D}$). However, the proposed refinement comes into play when normal refinement fails as the device to be reused is general and needs to be specialized dynamically. Hence, S-matching leads to dynamic refinement using an adapter.

One of the main ideas of dynamic reuse using the S-matching approach is that some behavior in $\mathcal{D}$ is hidden so that extra control paths not required in $\mathcal{F}$ will be removed. This hiding, however, is *state-based* unlike the global hiding operator in process algebras [11,21]. Consider, for example, $\mathcal{F}$ and $\mathcal{D}$ as shown in Fig. 13. In this example, $\mathcal{F}$ is a coffee vending machine that delivers coffee after a dollar coin is inserted. If $\mathcal{D}$ is an existing machine that delivers coffee only after two one dollar coins are inserted. If we want to reuse $\mathcal{D}$ to crate $\mathcal{F}$ then we need an adapter that waits for a coin to be inserted and subsequently forces the second coin (we might have another adapter that first forces a

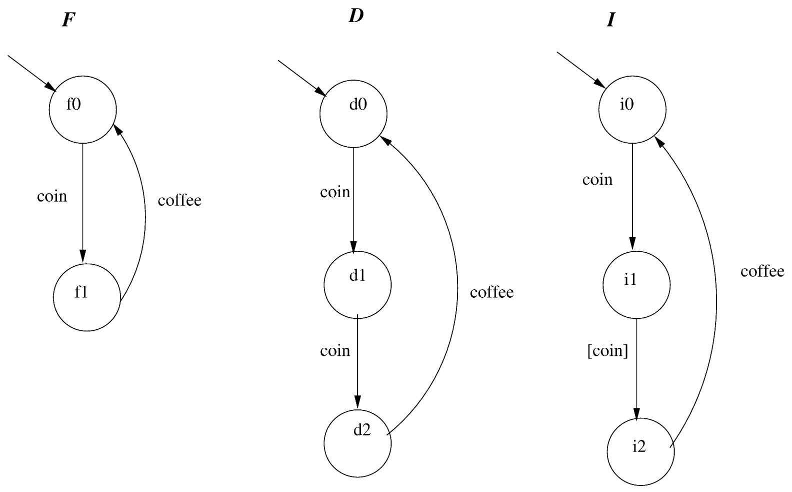


Fig. 13. State-based hiding.

coin before waiting for the second coin to be inserted). Thus, in the composite system, coin is visible in the first state while it is hidden in the next state. If we use hiding operator of process algebras, coin will be hidden globally, unlike the state-based hiding performed by our adapters.

The specification matching problem using S-matching looks superficially similar to the controllability problem [25] of discrete event systems (DES). In DES, a supervisor is synthesized to make a *plant* behave as the desired *specification*. The task of the supervisor is to disable certain actions of the plant at specific points. However, in DES control, the supervisor is not capable of state-based hiding which is induced in the composite system because of forcing actions of the adapter (which are not performed by the supervisor). Hence, for the reuse example in Fig. 13, no supervisors will be created.

7. Conclusion

The paper presents a new formal method for adaptive reuse in the domain of embed systems. Specifically, we have shown that: (1) a new automaton notation ALTS for precise specification of programmable devices; (2) a formal specification matching scheme to check the adaptability between two devices; (3) a specification matching tool implemented in tabled logic programming environment, which provides distinct advantages of rapid implementation; (4) a fully automated procedure to generate device adapter if there exists a specification matching relation; (5) a partial adapter for mismatching feedback; and (6) an application of the S-matching tool on intellectual property matching problems.

This work should be of interest to many communities involved in the research areas of computer science. First, for the formal methods and software engineering communities, specification matching provides a formal yet practical method for rapidly developing a reliable software for component adaptation. Second, for the logic programming community, this paper shows an attractive platform for encoding computational problems in both specification and verification of systems. And third, for the database community, specification matching can be used as a search key for component retrieval from a library of prefabricated components; attribute constraints in ALTS specification can make it possible to design a powerful and flexible search query scheme.

References

[1] M. Abadi, L. Lamport, The existence of refinement mappings, Theoretical Computer Science 82 (2) (1991) 253–284.
[2] S. Atkinson A formal model for integrated retrieval from software libraries, in: Proceeding of Technology of Object-Oriented Languages and Systems, 1996.
[3] F. Balarin, M. Chiodo, P. Guisto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanno-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara, Hardware Software Codesign of Embedded Systems—The POLIS Approach, Kluwer, 1997.
[4] S. Basu, M. Mukund, C.R. Ramakrishnan, I.V. Ramakrishnan, R.M. Verma, Local and symbolic bisimulation using tabled constraint logic programming, in: International Conference on Logic Programming, 2001, pp. 166–180.

[5] J. Bosch, Superimposition: A component adaptation technique, Information and Software Technology 41 (5) (1999).
[6] W. Chen, D.S. Warren, Tabled evaluation with delaying for general logic programs, Journal of the ACM 43 (1) (1996) 20–74.
[7] R. J. van Glabbeek, The linear time—branching time spectrum, in: International Conference on Concurrency Theory, 1990, pp. 278–297.
[8] E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 2000.
[9] B. Fischer, Specification-based browsing of software component libraries, in: Proc. 13th Automated Software Engineering, 1998, pp. 74–83.
[10] T.A. Henziger, Z. Manna, A. Pnueli, Temporal proof methodologies for real-time systems, in: Principles of Programming Languages, 1991, pp. 353–366.
[11] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall International, 1985.
[12] J. Jeng, B.H.C. Cheng, Specification matching for software reuse: A foundation, in: Proc. the ACM SIGSOFT Symposium on Software Reusability, 1995, pp. 97–105.
[13] J. Jussel, System-on-a-chip reuse platforms can dramatically shorten design cycles, Electronic Design 48 (21) (2000).
[14] V. Krishnan, S. Sethuraman, Reuse oriented slicing of specifications for embedded systems, Department of Electrical Engineering, Final Year Project Report, University of Auckland, 2002.
[15] J. Lloyd, Foundations of Logic Programming, Springer-Verlag, 1987.
[16] N.A. Lynch, M.R. Tuttle, An introduction to input/output automata, CWI Quarterly 2 (3) (1989) 219–246.
[17] N. Lynch, F. Vaandrager, Forward and backward simulations, part I: Untimed systems, Information and Computation 121 (2) (1995) 214–233.
[18] F. Maraninchi, N. Halbwachs, Compositional semantics of nondeterministic synchronous languages, in: Proceedings of the European Symposium on Programming (ESOP), 1996, pp. 235–249.
[19] K. Matzel, P. Schnorf, Dynamic component adaptation, Technical Report 97-6-1, Union Bank of Switzerland, 1997.
[20] M. Mezini, K. Lieberherr, Adaptive plug-and-play components for evolutionary software development, ACM Sigplan Notices 33 (10) (1998) 97–116.
[21] R. Milner, Communication and Concurrency, Prentice-Hall International, 1989.
[22] D.A. Peled, Software Reliability Methods, Springer-Verlag, 2001.
[23] G. Pemmasani, H.-F. Guo, Y. Dong, C.R. Ramakrishnan, I.V. Ramakrishnan, Online justification for tabled logic programs, in: International Symposium on Functional and Logic Programming, 2004, pp. 24–38.
[24] J. Penix, Automated component retrieval and adaptation using formal specifications, PhD Thesis, Univ. of Cincinnati, 1998.
[25] P.J.G. Ramadge, W.M. Wonham, The control of discrete event systems, Proceedings of the IEEE 77 (1989) 81–98.
[26] C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, D.S. Warren, LMC: A system for the specification and evaluation of logic-based model checking, ACM Software Engineering Notes 25 (1) (2000).
[27] E.J. Rollins, J.M. Wing, Specifications as search keys for software libraries, in: The International Conference on Logic Programming, MIT Press, 1991, pp. 173–187.
[28] P. Roop, Forced simulation: A formal approach to component based development of embedded systems, PhD Thesis, The Univ. of New South Wales, Sydney, 2000.
[29] P. S Roop, A. Sowmya, S. Ramesh, Forced simulation: A technique for automating component reuse in embedded systems, ACM Transactions on Design Automation of Electronic Systems 6 (4) (2001).
[30] P. S Roop, A. Sowmya, S. Ramesh, k-time forced simulation: A formal verification technique for IP reuse, in: IEEE International Conference on Computer Design, 2002, pp. 50–55.
[31] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, I. Bolsens, Hardware reuse at the behavioral level, in: Proceedings of the 36th Design Automation Conference, 1999, pp. 784–789.
[32] H. Tamaki, T. Sato, Old resolution with tabulation, in: International Conference on Logic Programming (ICLP), 1996, pp. 84–98.
[33] J.D. Ullman, J.E. Hopcroft, Introduction to Automated Theory Languages, and Computation, Addison-Wesley, 1990.
[34] M.Y. Vardi, Verification of concurrent programs: The automata-theoretic framework, in: IEEE Symposium on Logic in Computer Science, 1987, pp. 167–176.
[35] The XSB logic programming system v2.5, http://xsb.sourceforge.net/, 2003.
[36] A.M. Zarernski, J.M. Wing, Specification matching of software components, in: 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1995, pp. 6–17.