Specifying System Families with TLA^{+}

TATJANA KAPUS
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova ul. 17, SI-2000 Maribor
SLOVENIA
kapus@uni-mb.si

Abstract: In software product line engineering, families of similar products are developed instead of single ones. The systems produced as part of a family differ in the features they include. To ease the simultaneous design of multiple systems, methods to write a formal specification of a complete product line have been proposed. This paper shows how a TLA^{+} specification of a system family can be written, based on the representation of the family by a featured transition system. The specification of correctness properties of system families in TLA^{+} and their verification with the TLA^{+} model checker TLC are considered.

Key–Words: Software Product Line, Feature, Formal Specification, Temporal Logic of Actions, Featured Transition System, Safety, Liveness, Model Checking

1 Introduction

In *software product line engineering* (SPLE), several software systems, or *products*, are designed in parallel from a limited number of components (often called *features*) with different functionality, developed for the specific product domain. In this way, the development of the products can be more efficient as they normally have many features in common and the features can be reused [2]. SPLE is often used in the domain of safety-critical systems. At least for these, formal specification and verification have been recognised as important means for contributing to their correct behaviour. In order to be able to efficiently manage the common and distinctive features of the systems being developed within a *software product line* (SPL) and to take advantage of shared features in the verification process, there is a need for formal methods which enable the description of the behaviour of all the systems in one specification.

Originally, the well-known formal methods, such as process calculi and temporal logics, including the specification language TLA^{+}, have not been developed by having the specification of SPLs in mind. Only recently, some existing formal methods, such as modal transition systems, have been recognised as appropriate for the specification of SPLs, and new methods have been proposed specifically for SPLs [3, 1]. Among the latter, currently the best ellaborated seems to be the one by which a whole system family can be represented as a *featured transition system* (FTS) [2]. Besides proposing the FTS model, its authors also developed some languages for their description and tools for efficient automatic verification of system families by model checking branching- or linear-time temporal logic formulas.

TLA^{+} [5] is a formal specification language based on the Temporal Logic of Actions (TLA) [4] and ordinary mathematics, such as predicate logic and set theory, which makes it relatively easy to learn. Another favorable characteristic of TLA^{+} is that both the system specification, i.e. the description of its model, and the required correctness properties can be written in it. These and the fact that it has increasingly good verification tool support [9, 10] are the reasons that we have decided to explore how it could be used for the specification of SPLs. In this paper, we show that a system family can easily be specified in TLA^{+} if it is imagined as a kind of featured transition system.

In Sect. 2, we briefly explain TLA^{+}. In Sect. 3, the featured transition system model is presented informally with help of a popular SPL example from the literature—a family of coffee machines. The latter is also used in Sect. 4 for the explanation of the TLA^{+} specification of system families and their correctness properties. In Sect. 5, the possibility of using the TLA^{+} model checker TLC for system family verification is discussed. Section 6 concludes the paper.

2 TLA^{+}

In TLA^{+}, a system specification is written in the form of one or more modules. A module contains declarations of constants and variables as well as definitions of symbols representing the (parts of) formulas used in the specification. A module can be defined as an extension of some other modules, which means that the contents of the latter are imported into the former. One of the formulas is usually a canonical-form TLA formula, and it is in fact this one that specifies the possible system behaviours. Informally, in TLA$^+$ a behaviour is an infinite sequence of states, which are valuations of the declared variables of the system. The constants are like variables, but in contrast to the latter, it is a priori assumed that their values are fixed.

In this paper, we will need canonical-form TLA formulas of the form $I \wedge \Box[N]_v \wedge L$, where v represents variables of the system. A behaviour satisfies this formula iff predicate formula I is true in its initial state, every pair of consecutive states either satisfies action formula N or the value of variables from v is the same in both states, and the temporal formula L holds for the behaviour. N is usually a disjunction of action formulas. An action formula (in fact called an action in TLA$^+$) is like a predicate formula, except that it may contain primed variables. A primed variable, such as x', denotes the value of the variable in the next state. For example, a behaviour satisfies formula $x = 0 \wedge \Box[x < 10 \wedge x' = x + 1]_x$ iff x is 0 in its initial state and in every state, either x does not change or is smalller than 10 and incremented by 1. The temporal operators of TLA($^+$) are $\Box$ ("always"), $\Diamond$ meaning $\neg\Box\neg F$ ("eventually") and $\leadsto$ defined as $\Box(F \Rightarrow \Diamond G)$. In TLA($^+$), it must explicitly be specified if a variable does not change. Notation UNCHANGED v can be applied, and $[N]_v$ is short for $N \vee$ UNCHANGED v. L is usually a conjunction of fairness conditions for behaviours, such as $WF_v(A)$, which is short for a *weak fairness* condition. A behaviour satisfies it if either action $A \wedge (v' \neq v)$ is true in infinitely many consecutive state pairs or it is infinitely often disabled. An action is enabled in a state if it is possible to choose such a next state that it is true in this state pair. For example, action $x < 10 \wedge x' = x + 1$ is enabled in a state iff $x < 10$ holds in it.

Suppose that *Spec* is defined to denote a canonical TLA formula in a TLA$^+$ module and *Req* a temporal formula. In order to express that *Spec* is meant to be the behaviour specification of the system being specified and that *Req* is a required correctness property of the system, the module must contain a theorem statement such as THEOREM $\mathit{Spec} \Rightarrow \mathit{Req}$. Please, note that in TLA($^+$), satisfaction of a property by a system is expressed by implication. Besides the declaration of constants in a module, TLA$^+$ allows to write assumptions about their values in it by using the reserved word ASSUME . If, for example, the module with the above theorem contains some assumptions, then, informally, the module asserts: "If the constants of the system satisfy these assumptions, then the system satisfies the required property *Req*."

3 Featured Transition Systems

According to [2], a featured transition system is a transition system with an additional labelling function and a feature diagram (FD). A transition system consists of states and transitions between them. One or more states may be designated as the initial states. Each transition is labelled by an action (not a TLA action). States are labelled by sets of atomic propositions. A proposition is true in a state iff it is in the set of its labels. The graph in Fig. 2 without the parts of the labels on the edges starting by "/" represents a transition system (without state labels) with 14 states and, accidentally, as many actions (1EU, 1US, . . . , take_cup).

A feature diagram is the usual means for the representation of all the available features for an SPL and all the possible combinations of them in the products [8]. Basically, it consists of the set of features (nodes) and a decomposition relation between them [2]. One of the nodes is designated as the root node. The decomposition relation must form a tree or a directed acyclic graph with the root node. Figure 1 presents the feature diagram of the coffee machine product line as described in [1]. This diagram can be interpreted as follows. The Machine feature (denoted by m) is the root and must be present in all the products. If feature Machine is present, Sweet, Coin, and Beverage will also be as they are mandatory. The Ringtone feature may be present in the product or not, but if the Cappucino feature is included, so must be the Ringtone one. Feature Coffee must be included if Beverage is, whereas one, both, or none of features Cappucino and Tea may be supported. If feature Coin is present, either the Euro or Dollar feature must be present, but not both. Features Dollar and Cappucino must not be present in the same product. (Additional types of decompositions for FDs can be found in the literature.) If a FD is well-formed, then at least one product is possible. The reader can check that ten different coffee machines are allowed by the FD shown. One of them is, for instance, an "Italian coffee machine", which, like all the other machines, accepts coins, takes care for sugar in the beverages served, and can serve coffee, but accepts only euros and also serves cappucino, each beverage accompanied by a ringtone.

The graph in Fig. 2 and the FD in Fig. 1 together form the FTS of the coffee machine product line. The parts of the edge labels following the slash represent the above mentioned additional labelling function. These labels are propositional formulas,

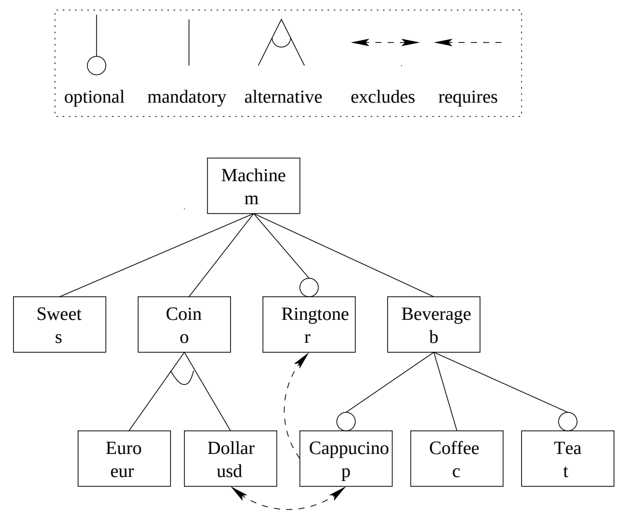

Figure 1: Feature diagram of coffee machine family [1].

in which propositions represent the features from the FD. For a given transition, the formula in its label indirectly denotes in which products—combinations of features—the transition is available. For example, the label $m \wedge \neg r$ on the transition with action skip in Fig. 2 means that this transition (the edge and its start and end states, of course) and thus action skip is available in all the products which contain feature Machine and do not contain feature Ringtone. It can also be seen that any product containing the latter will ring a tone after pouring a beverage.

In this way, the behaviour (i.e. the transition systems) of all the products allowed by the FD is specified by the FTS. For example, in Fig. 2 it can be seen that the Italian coffee machine first accepts one Euro, next the customer chooses if he (or she) wants sugar in the beverage, afterwards he chooses between coffee and cappucino, the chosen beverage is prepared, the tone rung, finally, the customer can take the beverage, and so on.

4 Specification of System Families with TLA+

The specification of a system family with TLA$^+$ can be based on its FTS representation. This approach is illustrated by module *CoffeeMachineSPL* in Figs. 3 and 4, which contains a TLA$^+$ specification of the coffee machine family described in Sect. 3. Variables have to be declared in order to specify the states and the transitions of the transition system underlying a FTS. In module *CoffeeMachineSPL*, variable st is used to represent the states of the transition system of Fig. 2 by different numbers, from 0 to 13. Variable ev

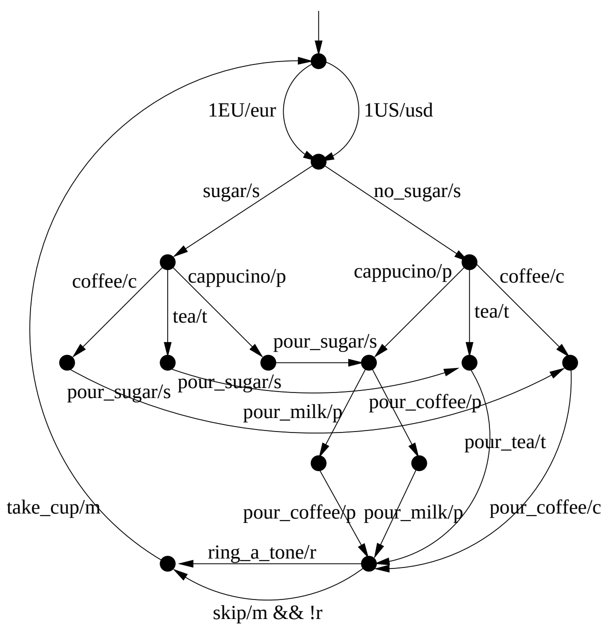

Figure 2: Featured transition system (with FD depicted in Fig. 1) of coffee machine family [1].

is used to represent the actions of the transitions. Symbol $Event$ denotes the set of possible actions. $Init$ defines the initial state of the transition system. If we had a FTS with a state-labelling function, the validity of its atomic propositions could naturally be specified by variables, possibly without the use of a variable such as st, which explicitly represents the states.

The action formulas named $Machine$, $Sweet$, ... describe the possible transitions of the transition system together with the additional labelling with propositional formulas, i.e. the transitions of the FTS. Although the propositional formulas in a FTS are used as transition labels, it is essential to notice that they in fact represent a part of the enabling condition of the transitions. In the action formulas, we, therefore, apply these labels as additional conjuncts. We represent the needed propositions ("features") by equally named (boolean) constants declared at the beginning of the module, in our case $m, s, o, \ldots$ The action formulas representing the FTS transitions can, of course, be organised in arbitrary groups of disjuncts. We decided to organise them with regard to the features to which they "belong". For example, if a transition has the labelling proposition formula m, we put it in the $Machine$ formula. Note that the skip transition is also specified in this formula, having the additional conjunct $\neg r$.

Having introduced the constants representing the features, we propose to specify the feature diagram in TLA$^+$ as an assumption about them. This is easy because it has been found that FDs can be described by

MODULE *CoffeeMachineSPL*

EXTENDS *Naturals*

CONSTANTS $m, s, o, r, b, eur, usd, p, c, t$

VARIABLES st, ev

$$\begin{array}{ll} boolfeat \triangleq & \land\, m \in \text{BOOLEAN} \land s \in \text{BOOLEAN} \\ & \land\, o \in \text{BOOLEAN} \land r \in \text{BOOLEAN} \\ & \land\, b \in \text{BOOLEAN} \land eur \in \text{BOOLEAN} \\ & \land\, usd \in \text{BOOLEAN} \land p \in \text{BOOLEAN} \\ & \land\, c \in \text{BOOLEAN} \land t \in \text{BOOLEAN} \end{array}$$

$$\begin{array}{ll} \text{ASSUME} & \land\, boolfeat \\ & \land\, m \land (m \equiv s \land o \land b) \land (r \Rightarrow m) \\ & \land\, (o \Rightarrow ((eur \land \neg usd) \lor (\neg eur \land usd))) \\ & \land\, (eur \lor usd) \Rightarrow o \\ & \land\, (b \equiv c) \land (p \Rightarrow b) \land (t \Rightarrow b) \\ & \land\, \neg(usd \land p) \land (p \Rightarrow r) \end{array}$$

$$vars \triangleq \langle st, ev \rangle$$

$$\begin{array}{ll} Event \triangleq & \{\text{“”}, \text{“1EU”}, \text{“1US”}, \text{“sugar”}, \text{“no\_sugar”}, \\ & \text{“coffee”}, \text{“tea”}, \text{“cappucino”}, \text{“pour\_sugar”}, \\ & \text{“pour\_coffee”}, \text{“pour\_tea”}, \text{“pour\_milk”}, \\ & \text{“cup\_taken”}, \text{“ring\_a\_tone”}, \text{“skip”}\} \end{array}$$

$$TypeInvariant \triangleq st \in 0..13 \land ev \in Event$$

$$\begin{array}{l} Machine \triangleq \\ \quad m \land \lor\, (st = 12) \land \neg r \land (ev' = \text{“skip”}) \land (st' = 13) \\ \qquad\quad \lor\, (st = 13) \land (ev' = \text{“cup\_taken”}) \land (st' = 0) \\ Sweet \triangleq \\ \quad s \land \lor\, (st = 1) \land (ev' = \text{“sugar”}) \land (st' = 2) \\ \qquad\ \lor\, (st = 1) \land (ev' = \text{“no\_sugar”}) \land (st' = 3) \\ \qquad\ \lor\, (st = 4) \land (ev' = \text{“pour\_sugar”}) \land (st' = 9) \\ \qquad\ \lor\, (st = 5) \land (ev' = \text{“pour\_sugar”}) \land (st' = 8) \\ \qquad\ \lor\, (st = 6) \land (ev' = \text{“pour\_sugar”}) \land (st' = 7) \\ Euro \triangleq \land\, eur \\ \qquad\quad \land\, (st = 0) \land (ev' = \text{“1EU”}) \land (st' = 1) \\ Dollar \triangleq \land\, usd \\ \qquad\quad \land\, (st = 0) \land (ev' = \text{“1US”}) \land (st' = 1) \\ Coffee \triangleq \\ \quad c \land \lor\, (st = 2) \land (ev' = \text{“coffee”}) \land (st' = 4) \\ \qquad\ \lor\, (st = 3) \land (ev' = \text{“coffee”}) \land (st' = 9) \\ \qquad\ \lor\, (st = 9) \land (ev' = \text{“pour\_coffee”}) \land (st' = 12) \\ Tea \triangleq \\ \quad t \land \lor\, (st = 2) \land (ev' = \text{“tea”}) \land (st' = 5) \\ \qquad\ \lor\, (st = 3) \land (ev' = \text{“tea”}) \land (st' = 8) \\ \qquad\ \lor\, (st = 8) \land (ev' = \text{“pour\_tea”}) \land (st' = 12) \\ Cappucino \triangleq \\ \quad p \land \lor\, (st = 2) \land (ev' = \text{“cappucino”}) \land (st' = 6) \\ \qquad\ \lor\, (st = 3) \land (ev' = \text{“cappucino”}) \land (st' = 7) \\ \qquad\ \lor\, (st = 7) \land (ev' = \text{“pour\_coffee”}) \land (st' = 11) \\ \qquad\ \lor\, (st = 11) \land (ev' = \text{“pour\_milk”}) \land (st' = 12) \\ \qquad\ \lor\, (st = 7) \land (ev' = \text{“pour\_milk”}) \land (st' = 10) \\ \qquad\ \lor\, (st = 10) \land (ev' = \text{“pour\_coffee”}) \land (st' = 12) \\ Ringtone \triangleq \\ \quad r \land (st = 12) \land (ev' = \text{“ring\_a\_tone”}) \land (st' = 13) \end{array}$$

Figure 3: TLA$^+$ module *CoffeeMachineSPL* (beginning)

$$\begin{array}{l} Init \triangleq (st = 0) \land (ev = \text{“”}) \\ Next \triangleq \lor\, Machine \lor Euro \lor Dollar \lor Sweet \\ \qquad\quad\ \lor\, Coffee \lor Tea \lor Cappucino \lor Ringtone \\ Fairness \triangleq WF_{vars}(Next) \\ Spec \triangleq Init \land \Box[Next]_{vars} \land Fairness \end{array}$$

$$\begin{array}{l} PropertyA \triangleq \lor\, \Box\Diamond(ev = \text{“1EU”}) \land \Box(ev \neq \text{“1US”}) \\ \qquad\qquad\quad\ \lor\, \Box\Diamond(ev = \text{“1US”}) \land \Box(ev \neq \text{“1EU”}) \\ PropertyB \triangleq usd \Rightarrow \Box(ev \neq \text{“cappucino”}) \\ PropertyC \triangleq \\ \quad p \Rightarrow (ev \in \{\text{“coffee”}, \text{“tea”}, \text{“cappucino”}\} \\ \qquad\qquad \leadsto ev = \text{“ring\_a\_tone”}) \\ PropertyD \triangleq ev = \text{“coffee”} \leadsto ev = \text{“pour\_coffee”} \\ \text{THEOREM } Spec \Rightarrow TypeInvariant \land PropertyA \land \ldots \end{array}$$

Figure 4: TLA$^+$ module *CoffeeMachineSPL* (end)

propositional logic. Such a description can, of course, be written in several equivalent forms. The safest way is to use the straightforward rules for describing different decomposition types (e.g. [11]), as in module *CoffeeMachineSPL* beginning in Fig. 3.

Action labels of the transitions can be represented in different ways in a TLA$^+$ specification, depending on the kinds of system properties wanted to be expressed. If interested in liveness properties, it is important to take care that executing a transition differs from an idle step in the specification [7]. For the example in this paper, it suffices to represent the action labels by assigning the appropriate action to ev in the next state of the specified transitions. We introduced a "null" action "" in *Event* so as to be used as the initial value of ev.

Formally, it is the theorem statement in Fig. 4 that indicates that formula *Spec* in the same figure is meant to be the behavioural specification of our system (in our case the system family). By the semantics of ASSUME and THEOREM mentioned in Sect. 2, in our case, it is the specification of all the coffee machines in which the constants representing the features fulfill the assumption specifying the feature diagram. We decided to include weak fairness condition *Fairness* in *Spec* because it is realistic to assume that the coffee machines as well as their customers eventually do something if possible (e.g. a coffee machine serves a chosen beverage and a customer takes it).

In Fig. 4, we give examples of the specification of correctness properties expected either of all the coffee machines of the family or of some of them. If a property specification does not include any "feature" constant, then it is a specification for the whole product family. We tried to specify the properties analogous to the similarly named ones in [1]. *PropertyA* requires every coffee machine to accept either dollar or euro coins, but not both. $PropertyB$ asserts that American coffee machines do not offer cappucino. $PropertyC$ asserts that if a coffee machine offers cappucino, then it rings a tone for every beverage served. $PropertyD$ asserts that once a coffee has been selected, the coffee is eventually delivered. Since TLA($^+$) does not contain operators "until" and "unless", the properties such as "a coffee machine may never deliver a coffee before a coin has been inserted" [1] cannot precisely be specified in TLA$^+$ without the use of so-called hidden variables [6]. If these are used, a so-called refinement mapping has to be found to subsequently be able to carry out the verification automatically.

5 Verification of System Families with Model Checker TLC

TLC is a tool for automatic verification of finite-state TLA$^+$ specifications by model checking, developed by the author of TLA$^+$ and colleagues. It is also built in the TLA Toolbox development environment [9], but for the purpose of this paper, we used it without the latter. We used TLC2 version 2.03 [10].

The form of TLA$^+$ specification of system families presented in Sect. 4 is not appropriate for model checking by using TLC because this tool requires that every constant declared in the modules to be verified is instantiated by a concrete value [5]. In this way, only one member of a product family could be verified in one TLC run. For example, we could verify the Italian coffee machine by setting constants m, s, o, b, c, eur, p and r of module *CoffeeMachineSPL* beginning in Fig. 3 to TRUE and the rest to FALSE in the TLC configuration file, which has to be prepared before the verification.

It is desirable for a model checker to verify all the products of a SPL in one run [2]. For that purpose, the feature propositions have to be represented by variables instead of constants in the TLA$^+$ specification, and these variables must be specified not to be changed by any transition. (A similar approach is presented in [2] to use the existing model checker NuSMV for FTS verification.) The feature diagram has to be specified by propositions in the formula that specifies the initial state(s) of the system (family), and not in an assumption. Figure 5 shows the *CoffeeMachineSPL* module with the necessary changes. Of course, action $unch$ has to be included in all the action formulas that specify the transitions, or equivalently, it could be added as a conjunct to $Next$. The reader can notice that in Fig. 5, in $Init$, we present an obvious simplification of the feature diagram description from the assumption of Fig. 3. There is no need

MODULE *CoffeeMachineSPL*

EXTENDS $Naturals$
VARIABLES $m, s, o, r, b, eur, usd, p, c, t, st, ev$
$boolfeat \triangleq \ldots$

$vars \triangleq \langle m, s, o, r, b, eur, usd, p, c, t, st, ev \rangle$
$Event \triangleq \ldots$
$unch \triangleq$ UNCHANGED $\langle m, s, o, r, b, eur, usd, p, c, t \rangle$
$TypeInvariant \triangleq boolfeat \land st \in 0..13 \land ev \in Event$

$Machine \triangleq$
$\quad \land\, m \land \lor\, (st = 12) \land \neg r \land (ev' = \text{"skip"}) \land (st' = 13)$
$\qquad\quad \lor\, (st = 13) \land (ev' = \text{"cup\_taken"}) \land (st' = 0)$
$\quad \land\, unch$
$\ldots$
$Init \triangleq \land\, boolfeat$
$\qquad\quad \land\, m \land s \land o \land b \land c$
$\qquad\quad \land\, (eur \land \neg usd) \lor (\neg eur \land usd)$
$\qquad\quad \land\, \neg(usd \land p) \land (p \Rightarrow r)$
$\qquad\quad \land\, (st = 0) \land (ev = \text{""})$
$\ldots$

Figure 5: Module *CoffeeMachineSPL* for TLC

to write theorems in TLA$^+$ modules for TLC because the system specification and properties to be verified are identified in the TLC configuration file. We have run TLC on the changed module. The satisfaction of all the properties was confirmed in less than a second. 170 states were generated. As expected, ten different initial states were generated, each representing the initial state of one product. It should be noticed that since TLC is an explicit model checker, the verification of a FTS in one run does not contribute to space savings compared to the group of runs for individual products. Because the feature information is represented by variables, it is part of all the generated states, and therefore, no products have any states in common.

Ideally, if a model checker discovers that a required property is not satisfied by the product family being verified, it should indicate the products which do not satisfy the property in the form of a simple propositional formula indicating the features that are responsible for the violation [2]. Normally, TLC stops running when it discovers that a property is not satisfied and prints a (part of) behaviour which violates it. From this, the user can read the values of the variables and thus the feature combination (i.e. the product) for which the property does not hold. TLC always first generates the reachable states and for every state generated checks the satisfaction of all the required invariants. It is possible to run TLC by using option `-continue`. In this case, when TLC discovers that an invariant is violated, it prints its name and an error trace, but continues the verification in this way until it generates the complete state space. It checks other

$$
\begin{array}{l}
\textsc{assume} \\
\quad \forall\, M, S, O, R, B, EUR, USD, P, C, T \in BOOLEAN\ : \\
\qquad (M \land (M \equiv S \land O \land B) \land (R \Rightarrow M) \\
\qquad \land(O \Rightarrow ((EUR \land \neg USD) \lor (\neg EUR \land USD))) \\
\qquad \land((EUR \lor USD) \Rightarrow O) \\
\qquad \land(B \equiv C) \land (P \Rightarrow B) \land (T \Rightarrow B) \\
\qquad \land\neg(USD \land P) \land (P \Rightarrow R)) \\
\quad\ \equiv (M \land S \land O \land B \land C \\
\qquad \land((EUR \land \neg USD \land P \land T \land R) \\
\qquad \lor(EUR \land \neg USD \land P \land \neg T \land R) \\
\qquad \lor(EUR \land \neg USD \land \neg P \land \neg T \land R) \\
\qquad \lor(EUR \land \neg USD \land \neg P \land T \land R) \\
\qquad \lor(EUR \land \neg USD \land \neg P \land \neg T \land \neg R) \\
\qquad \lor(EUR \land \neg USD \land \neg P \land T \land \neg R) \\
\qquad \lor(\neg EUR \land USD \land \neg P \land \neg T \land R) \\
\qquad \lor(\neg EUR \land USD \land \neg P \land T \land R) \\
\qquad \lor(\neg EUR \land USD \land \neg P \land \neg T \land \neg R) \\
\qquad \lor(\neg EUR \land USD \land \neg P \land T \land \neg R)))
\end{array}
$$

Figure 6: Possible products confirmed by TLC

kinds of the required properties afterwards, but stops running if it discovers a violation. It does print an error trace, but no property name. From the printed error traces, the user can, of course, read all the feature combinations that caused them.

If we would like to verify a property for a subset of products and the state space of the FTS is very large, it might help to specify it in a different way and verify it separately. Notice that such properties have the form $P \Rightarrow Req$ where propositional formula P specifies the subset of products. We can add P as a conjunct to the family specification and write Req as the required property. TLC will generate the state space for the product subset and verify Req in it. For example, if we move p of $PropertyC$ defined in Fig. 4 to $Init$, two initial states are generated, representing the two products offering cappucino, and the property is verified for them by generating only 50 states.

TLC can also be applied to verify if a propositional formula used to specify a feature diagram specifies exactly the desired feature combinations. This can be done by putting such an assertion in an assumption of a TLA$^+$ module because TLC always first checks the validity of all the assumptions. For example, we applied TLC to check that the coffee machine family contains products with ten different feature combinations by writing the assumption shown in Fig. 6. The ten combinations are evident from the right side of operator $\equiv$.

6 Conclusion

We showed that system families can be specified with TLA$^+$ in such a way that the TLC model checker can be applied for their verification. In the future, the `-continue` option could be adapted for SPLs so that it would be aware of the products it checks and concisely indicate the violating products. It should be noticed that the number of possible products is exponential in the number of available features and that the latter is often much larger in practice than in this paper [2]. It would be necessary to introduce symbolic state space representation and manipulation in TLC, not only for the sake of SPL verification, but to increase the practical applicability of TLC in general.

References:

[1] H. Asirelli, M. H. ter Beek, S. Gnesi and A. Fantechi, Formal Description of Variability in Product Families, in E. Almeida et al. (eds.), *Proceedings 15th Int. Software Product Line Conference (SPLC 2011)*, IEEE Computer Society, Los Alamitos, 2011, pp. 130–139.

[2] A. Classen, Modelling and Model Checking Variability-Intensive Systems, PhD Thesis, PReCISE Research Centre, Faculty of Computer Science, University of Namur, October 2011.

[3] D. Fischbein, S. Uchitel and V. Braberman, A foundation for behavioural conformance in software product line architectures, in *Proceedings ISSTA 2006 workshop on Role of software architecture for testing and analysis (ROSATEA 06)*, ACM Press, New York, 2006, pp. 39–48.

[4] L. Lamport, The temporal logic of actions, *ACM Trans. Prog. Lang. Syst.* 16, 1994, pp. 872–943.

[5] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley Professional, 2002, available at: http://research.microsoft.com/en-us/um/people/lamport/tla/book.html.

[6] S. Merz, On the logic of TLA$^+$, *Comput. Inform.* 22, 2003, pp. 351–379.

[7] M. Reynolds, Changing nothing is sometimes doing something: fairness in extensional semantics, Technical Report tr-98-02, King's College, 1998.

[8] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux and Y. Bontemps, Generic Semantics of Feature Diagrams, *Comp. Netw.* 51, 2007, pp. 456–479.

[9] TLA Toolbox, http://www.tlaplus.net/tools/tla-toolbox/.

[10] TLA+ Tools, http://research.microsoft.com/en-us/um/people/lamport/tla/tools.html.

[11] J. White, D. Benavides, D. C. Schmidt, P. Trinidad, B. Dougherty and A. Ruiz-Cortes, Automated Diagnosis of Feature Model Configurations, *J. Syst. Soft.* 83, 2010, pp. 1094–1107.