AspectLTL: An Aspect Language for LTL Specifications [*]

[draft version]

Shahar Maoz
RWTH Aachen University, Germany
maoz@se-rwth.de

Yaniv Sa'ar
Weizmann Institute of Science, Israel
yaniv.saar@weizmann.ac.il

ABSTRACT

We present *AspectLTL*, a temporal-logic based language for the specification and implementation of crosscutting concerns. AspectLTL enables the modular declarative specification of expressive concerns, covering the addition of new behaviors, as well as the specification of safety and liveness properties. Moreover, given an AspectLTL specification, consisting of a base system and a set of aspects, we provide AspectLTL with a composition and synthesis-based weaving process, whose output is a correct-by-construction executable artifact. The language is supported by a prototype tool and is demonstrated using a running example.

1. INTRODUCTION

A common characteristic of languages of the aspect-oriented programming paradigm [19] and of related advanced modularity paradigms, is that their structural building blocks specify separate, yet possibly inter-dependent crosscutting concerns. This poses a major challenge, which is the automated composition or weaving of these separate specifications or program pieces into a single correct implementation, one which can be programmatically executed and indeed supports and follows the different concerns. To address this challenge, a balance needs to be made between the extent of modularity of the program specification or code to go beyond traditional abstraction boundaries, the language's expressive power in specifying and manipulating system behavior, and the ability to automatically transform or weave such a modular specification or code into an executable correct artifact. The more modular or structurally and syntactically separate yet semantically inter-dependent and expressive the language constructs are, the more difficult it becomes to automatically generate a correct implementation.

In this work we present *AspectLTL*, a language for the *specification and implementation* of crosscutting concerns, based on linear temporal logic (LTL) [26]. The aspects of AspectLTL, called LTL aspects, enable the declarative specification of expressive crosscutting concerns. These include the specification of safety properties, which may be used to prevent a base system from visiting 'bad states', the specification of liveness (fairness) properties, which may be used to force a base system to visit 'good states' (infinitely often), and the addition of new behaviors to a base system, which is done by specifying the existence of new transitions and new states as necessary. To use the categorization by Katz [16], LTL aspects can specify spectative, regulative, and invasive aspects.

Moreover, we provide AspectLTL with a synthesis-based weaving process, whose output is a correct-by-construction executable artifact. Following a composition of the specified aspects with a base system, using symbolic disjunctive and conjunctive operations, we formulate the problem of correct weaving as a synthesis problem [27], essentially a game between the environment and the (augmented) base system. An algorithm based on [25] is used to solve the game, that is, to provide the augmented system with a winning strategy, if any.

If a winning strategy is found, it is presented as a deterministic, executable automaton, which represents an augmented base system whose behavior is guaranteed to adhere to the specified aspects, in all possible environments. If a winning strategy is not found, we know that it does not exist, that is, that no system exists which is based on the base system and can adhere to the specified LTL aspects in all environments. Thus, LTL aspect composition and synthesis is sound and complete.

Our work can be viewed as demonstrating how correct aspect weaving can be reduced to (a variant of) the classical synthesis problem. The synthesis problem was first identified by Church [5], and several methods have been proposed for its solution [4, 30]. The problem was considered again in [27] in the context of synthesizing reactive modules from a specification given in LTL, but the high complexity established in [27, 28] caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners from ever attempting to use it for any sizeable system

[*]This research was supported by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science. In addition, part of this research was funded by an Advanced Research Grant awarded to David Harel of the Weizmann Institute from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013). Finally, the first listed author acknowledges support from a postdoctoral Minerva Fellowship, funded by the German Federal Ministry for Education and Research.

development. However, a more recent work [25] presents an efficient polynomial time algorithm for the synthesis of specifications of a subset of LTL, namely the class of *Generalized Reactivity(1)* formulae (GR(1)). Our work uses (a fragment of) the GR(1) fragment of LTL and the algorithm presented in [25]. We give a brief overview of LTL and of the work in [25] in Sec. 2.

An AspectLTL specification is made of a base system, given as a finite-state machine specified in an SMV [1] format, and a set of LTL aspects, each of which is specified in a similar SMV-like format, containing a symbolic representation of the aspect's added behaviors (transitions) and a related LTL specification. As the base system assumes nothing about the LTL aspects it may be woven with, AspectLTL supports *obliviousness*. Moreover, the use of the symbolic representation provides *quantification*: rather than relating to concrete states, a single formula typically relates to a set of states. These two language features, obliviousness and quantification, are considered a distinguishing characteristic of aspect languages [11], and so, indeed, AspectLTL is an aspect language.

An aspect language has a *join point model* (JPM), which defines the points where an aspect may interfere with a base, how these points may be specified, and how the additional aspect behavior is defined. AspectLTL features a very general and permissive JPM: it allows new transitions to be added at any state of the base system: all states are possible *join points* and *pointcuts* are not specified explicitly. The "advice" of LTL aspects has not only local and specific effect on selected points along the execution, but also a global, temporal and general effect on ongoing, infinite executions.

Some previous works have formally characterized aspects using LTL or automata, mainly in order to prove aspect correctness using model-checking techniques (see, e.g., [12, 21]). Other works translate LTL properties into corresponding monitors written using aspect code (e.g., in AspectJ), as a means for LTL runtime verification (see, e.g., [32]). In contrast to the above two lines of work, our main goal is to use an LTL characterization of aspects as an input for a composition and synthesis process, in order to produce a correct-by-construction executable system. In other words, we use LTL not only as a specification language but also as a programming language, leading directly into an executable artifact.

An interesting line of research that has attracted attention in recent years is the correct composition and program synthesis of features. Typically, these works consider composition at the level of safe type checking (see, e.g., [14, 33]): features can be composed if the resulting program is type-safe and compiles. AspectLTL program synthesis is different: in addition to type safety, correct composition is defined at the level of the semantics of the specification. We discuss related work in Sec. 9.

To demonstrate our ideas in this paper, we use a running example. The example is initially built from an underlying base system, which models a service for students exams. We start off with a base system, and use it as a minimal basis on which to add aspects. The running example is small enough to fit into a paper, yet complex enough to allow us to highlight the unique features of our work.

AspectLTL is supported by a prototype Eclipse plug-in, which we have developed on top of JTLV [29], a framework for the development of verification algorithms, using BDD-based symbolic mechanisms. We used this prototype to define several AspectLTL specifications, to weave them, and to run the generated executable artifacts. We briefly describe the plug-in's features in Sec. 8.

We consider the version of AspectLTL presented in this paper to be a basic version, capturing the essence of the language. Extensions to its expressive power and some other additional features are briefly discussed in Sec. 6. Several possible applications of our work are discussed in Sec. 7.

The paper is organized as follows. In the next section we briefly provide background material on LTL and synthesis. Sec. 3 defines the basic AspectLTL language and Sec. 4 defines its weaving mechanism, together with some examples. Sec. 5 presents additional examples of LTL aspects. Sec. 6 discusses possible extensions to the basic version of AspectLTL presented in the previous sections, and Sec. 7 discusses some advanced topics and future work related to the language and its possible applications. Sec. 8 presents the prototype implementation, Sec. 9 discusses related work, and Sec. 10 concludes and suggests directions for future work.

2. PRELIMINARIES

We give a rather technical but short introduction to linear temporal logic, to the synthesis problem, and to its solution in [25] as used in this paper.[1]

2.1 Linear Temporal Logic

Linear temporal logic (LTL) [22, 26] extends propositional logic with operators that describe variables valuations along infinite computation paths. Given a finite set of atomic propositions P, LTL formulae are constructed as

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi \mathcal{U} \varphi \mid \circleddash \varphi \mid \varphi \mathcal{S} \varphi$$

where $\bigcirc \varphi$ is the *next* temporal operator, roughly meaning that φ is true in the next step in the computation, $\varphi \mathcal{U} \psi$ is the *until* operator, roughly meaning that in any sequence of future steps φ is true *until* ψ is true, $\circleddash \varphi$ is the *previous* operator, the past version of the *next* operator $\bigcirc \varphi$, and $\varphi \mathcal{S} \psi$ is the *since* operator, the past version of *until*. We use the usual abbreviations of the Boolean connectives $\wedge$, $\rightarrow$ and $\leftrightarrow$ and the usual definitions for true and false. Additional future temporal operators, $\diamondsuit$ (*eventually*) and $\square$ (*globally*), are defined as abbreviations to $\mathsf{true}\mathcal{U}\varphi$ and $\neg \diamondsuit \neg\varphi$, respectively. Additional past temporal operators, $\diamondminus$ (*once*), $\boxminus$ (*historically*), and $\mathcal{B}$ (*backto*), are defined as abbreviations to $\mathsf{true}\mathcal{S}\varphi$, $\neg \diamondminus \neg\varphi$, and $(\varphi \mathcal{S} \psi) \vee \boxminus \varphi$, respectively.

A model σ for a formula φ is an infinite sequence of truth assignments to propositions. Namely, if $\widehat{P}$ is the set of propositions appearing in φ, then for every finite set P such that $\widehat{P} \subseteq P$, a word in $(2^P)^\omega$ is a model. Given a model $\sigma = \sigma_0, \sigma_1, \ldots$, we denote by σ_i the set of propositions at position i. For a formula φ and a position $i \geq 0$, we say that φ *holds at position* i *of* σ, written $\sigma, i \models \varphi$, and define it inductively as follows:

- For $p \in P$ we have $\sigma, i \models p$ iff $p \in \sigma_i$.
- $\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$

[1]In our work we use variables that range over any finite domain. These can be reduced to the Boolean variables used in the theoretical framework below.

- $\sigma, i \models \varphi \vee \psi$ iff $\sigma, i \models \varphi$ or $\sigma, i \models \psi$
- $\sigma, i \models \bigcirc \varphi$ iff $\sigma, i+1 \models \varphi$
- $\sigma, i \models \varphi \mathcal{U} \psi$ iff there exists $k \geq i$ such that $\sigma, k \models \psi$ and $\sigma, j \models \varphi$ for all j, $i \leq j < k$
- $\sigma, i \models \ominus \varphi$ iff $i > 0$ and $\sigma, i-1 \models \varphi$
- $\sigma, i \models \varphi \mathcal{S} \psi$ iff there exists $k, 0 \leq k \leq i$ such that $\sigma, k \models \psi$ and $\sigma, j \models \varphi$ for all j, $k < j \leq i$

If $\sigma, 0 \models \varphi$, then we say that φ *holds* on σ and denote it by $\sigma \models \varphi$. A set of models M satisfies φ, denoted $M \models \varphi$, if every model in M satisfies φ.

A formula that does not include temporal operators is a *Boolean formula* (or an *assertion*). Given a set of Boolean variables $\mathcal{V}$, we define a *state* s to be an interpretation of $\mathcal{V}$, assigning to each variable $v \in \mathcal{V}$ a value $s[v] \in \{0, 1\}$. We denote the set of all possible states by $\Sigma_{\mathcal{V}}$ (or simply Σ, if $\mathcal{V}$ is clear from the context), i.e., $\Sigma_{\mathcal{V}} = 2^{\mathcal{V}}$. We extend the evaluation function $s[\cdot]$ to Boolean expressions over $\mathcal{V}$ in the usual way. A state s *satisfies* an assertion φ, denoted by $s \models \varphi$, iff $s[\varphi] = \mathsf{true}$. We say that s is a φ-state if $s \models \varphi$. Given a formula φ and a set of states $S \subseteq \Sigma_{\mathcal{V}}$, we say that S satisfies φ, denoted by $S \models \varphi$, if $s \models \varphi$ holds for all $s \in S$.

2.2 Discrete Systems

A *discrete system* (DS) [17] is a symbolic representation of a transition system with finitely many states. Formally, a DS $\mathcal{D} = \langle \mathcal{V}, \theta, \rho \rangle$ consists of the following components:

- $\mathcal{V} = \{v_1, ..., v_n\}$: A finite set of Boolean variables. A *state* s is an interpretation of $\mathcal{V}$, i.e., $s \in \Sigma_{\mathcal{V}}$.
- θ : The *initial condition*. This is an assertion over $\mathcal{V}$ characterizing all the initial states of the DS. A state is called *initial* if it satisfies θ.
- ρ : A *transition relation*. This is an assertion $\rho(\mathcal{V} \cup \mathcal{V}')$, relating a state $s \in \Sigma_{\mathcal{V}}$ to its $\mathcal{D}$-successors $s' \in \Sigma_{\mathcal{V}'}$, i.e., $(s, s') \models \rho$.

We define a *run* of the DS $\mathcal{D}$ to be a maximal sequence of states $\sigma = s_0, s_1, \ldots$ satisfying (i) *initiality*, i.e., $s_0 \models \theta$, and (ii) *consecution*, i.e., for every $j \geq 0$, $(s_j, s'_{j+1}) \models \rho$. A sequence σ is maximal if either σ is infinite or $\sigma = s_0, \ldots, s_k$ and s_k has no $\mathcal{D}$-successor, i.e., for all $s_{k+1} \in \Sigma$, $(s_k, s'_{k+1}) \not\models \rho$.

We say that a DS $\mathcal{D}$ *satisfies* a specification φ, denoted $\mathcal{D} \models \varphi$, if every run of $\mathcal{D}$ satisfies φ.

Given a subset of variables $\mathcal{X} \subseteq \mathcal{V}$, a DS $\mathcal{D}$ is *deterministic with respect to* $\mathcal{X}$, if (i) for all states $s, t \in \Sigma_{\mathcal{V}}$, if $s \models \theta$, $t \models \theta$, and both s and t have the same projection to the variables in $\mathcal{X}$, then $s = t$, and (ii) for all states $s, s', s'' \in \Sigma_{\mathcal{V}}$, if $(s, s') \models \rho$, $(s, s'') \models \rho$, and both s' and s'' have the same projection to the variables in $\mathcal{X}$, then $s' = s''$. Otherwise, $\mathcal{D}$ is called *non-deterministic*. Note that conventional programs (i.e., "real" programs) are deterministic with respect to their input variables.

Given a subset of variables $\mathcal{X} \subseteq \mathcal{V}$, a DS $\mathcal{D}$ is *complete with respect to* $\mathcal{X}$, if (i) for every assignment $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$, there exists a state $s \in \Sigma_{\mathcal{V}}$ such that its projection to $\mathcal{X}$ is $s_{\mathcal{X}}$, and $s \models \theta$, and (ii) for all states $s \in \Sigma_{\mathcal{V}}$ and assignments $s'_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$, there exists a state $s' \in \Sigma_{\mathcal{V}}$ such that its projection to $\mathcal{X}$ is $s'_{\mathcal{X}}$, and $(s, s') \models \rho$.

A deterministic and complete discrete system is called a *controller*.

Finally, we are interested in *open* systems, that is, systems that interact with their environment. We model an open system by a discrete system whose variables are divided between environment controlled variables (inputs) and system controlled variables (outputs). A specification for an open system is intended to hold for all possible environments. That is, to satisfy a specification, the system should guarantee that all its runs satisfy the specification, regardless of the environments choice of assignments to input variables.

2.3 Game Structures

We define a *game structure* (GS) $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi \rangle$ with the following components:

- $\mathcal{V} = \{v_1, \ldots, v_n\}$: A finite set of Boolean variables. A state and the set of states $\Sigma_{\mathcal{V}}$ are defined as before.
- $\mathcal{X} \subseteq \mathcal{V}$ is a set of *input variables*. These are variables controlled by the environment.
- $\mathcal{Y} = \mathcal{V} \setminus \mathcal{X}$ is a set of *output variables*. These are variables controlled by the system.
- θ_e is an assertion over $\mathcal{X}$ characterizing the initial states of the environment.
- θ_s is an assertion over $\mathcal{V}$ characterizing the initial states of the system.
- $\rho_e(\mathcal{V}, \mathcal{X}')$ is the transition relation of the environment. This is an assertion relating a state $s \in \Sigma$ to a possible next input value $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ by referring to unprimed copies of $\mathcal{V}$ and primed copies of $\mathcal{X}$. The transition relation ρ_e identifies valuation $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ as a *possible input* in state s if $(s, s_{\mathcal{X}}) \models \rho_e$.
- $\rho_s(\mathcal{V}, \mathcal{X}', \mathcal{Y}')$ is the transition relation of the system. This is an assertion relating a state $s \in \Sigma$ and an input value $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ to an output value $s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ by referring to primed and unprimed copies of $\mathcal{V}$. The transition relation ρ_s identifies a valuation $s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ as a *possible output* in state s reading input $s_{\mathcal{X}}$ if $(s, s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \rho_s$.
- φ is the winning condition, given by an LTL formula.

For two states s and s' of G, s' is a *successor* of s if $(s, s') \models \rho_e \wedge \rho_s$. A *play* σ of G is a maximal sequence of states $\sigma : s_0, s_1, \ldots$ satisfying *initiality* namely $s \models \theta_e \wedge \theta_s$, and *consecution* namely, for each $j \geq 0$, s_{j+1} is a successor of s_j. Let G be a GS and σ be a play of G. From a state s, the environment chooses an input $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ such that $(s, s_{\mathcal{X}}) \models \rho_e$ and the system chooses an output $s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ such that $(s, s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \rho_s$.

A play σ is *winning for the system* if it is infinite and it satisfies φ. Otherwise, σ is *winning for the environment*.

A *strategy* for the system is roughly defined to be a partial function $f : \Sigma^+ \times \Sigma_{\mathcal{X}} \mapsto \Sigma_{\mathcal{Y}}$, such that given a sequence of states and an input valuation, ranges to an output valuation. Strategy f is *winning* for the system from state $s \in \Sigma_{\mathcal{V}}$ if all plays departing from s are winning for the system. A state $s \in \Sigma_{\mathcal{V}}$ is a *winning state* for the system if there exists a winning strategy for the system from s. For player environment, strategies, winning strategies, and winning states are defined dually.

A game structure G is said to be *winning* for the system, if for all $s_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$, if $s_{\mathcal{X}} \models \theta_e$, then there exists $s_{\mathcal{Y}} \in \Sigma_{\mathcal{Y}}$ such that $(s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \theta_s$ and $(s_{\mathcal{X}}, s_{\mathcal{Y}})$ is a winning state, namely there exists a winning stategy departing from state $(s_{\mathcal{X}}, s_{\mathcal{Y}})$.

2.4 Realizability and Synthesis

Given a specification, *realizability* amounts to checking whether there exists a controller that satisfies it. If the specification is realizable, then the construction of such a controller constitutes a solution for the *synthesis* problem.

In our current work, we are interested in the synthesis of LTL specifications that are written (or can be rewritten) in the form defined below.

Definition 1.
Let $\mathcal{X}$ be a set of input variables, and $\mathcal{Y}$ be a set of output variables. We define the following fragment of LTL *formulae*[2] *of the form*

$$\varphi : \varphi_i \ \wedge \ \varphi_t \ \wedge \ \varphi_g \tag{1}$$

where

(i) φ_i is a Boolean formula which characterizes the initial states of the implementation.

(ii) φ_t is a formula of the form $\bigwedge_{i \in I} \Box B_i$ where each B_i is a Boolean combination of variables from $\mathcal{X} \cup \mathcal{Y}$ and expressions of the form $\bigcirc v$ where $v \in \mathcal{X} \cup \mathcal{Y}$. φ_t characterizes the transition relation of the implementation.

(iii) φ_g is a formula of the form $\bigwedge_{i \in I} \Box \Diamond B_i$ where each B_i is a Boolean formula. φ_g characterizes liveness requirements for the implementation.

[25] presented an efficient polynomial time algorithm for the realizability and synthesis of specifications of the class of *Generalized Reactivity(1)* formulae (GR(1)). The GR(1) fragment contains formulas of the form defined in Equ. 1 and thus the solution presented in [25] is good for our needs.

The solution for the problem of synthesis of GR(1) specifications is based on a reduction to a game structure played between a system and an environment. The goal of the system is to satisfy the specification regardless of the actions of the environment (that is, regardless of the choice of values the environment assigns to the input variables). The game solution uses a μ-calculus formulation that characterizes the set of winning states for the system player and can be computed using a symbolic BDD-based polynomial (quadratic) algorithm, that is an implementation of the nested fixpoint in the μ-calculus formulation (see [10, 25]).

If for every initial assignment to environment variables, there exists a winning strategy for the system player, we know that the specification is realizable. If the specification is realizable, the game solution can be used to produce a correct-by-construction controller, which follows a winning strategy.

Finally, as noted by [25] and shown in case studies [2, 3], many specifications written in practice can be rewritten to the GR(1) format, including response properties of the form $\Box(p \rightarrow \Diamond q)$, the past fragment of LTL, and many additional interesting cases.

Our work uses a fragment of the GR(1) fragment of LTL and the synthesis algorithm given in [25].

[2]also known as the temporal semantics of *just discrete systems* (JDS) [17].

3. LANGUAGE DEFINITION

We are now ready to present the AspectLTL language. We define the abstract syntax, the concrete syntax, and the semantics of AspectLTL. The ideas and definitions are demonstrated using a running example made of a base system and number of aspects.

3.1 Base System

Definition 2 (Base system).
A base system is a discrete system $\mathcal{B} = \langle \mathcal{V}_B = \mathcal{V}_B^e \cup \mathcal{V}_B^s, \theta_B, \rho_B \rangle$ consisting of the following components:

- *$\mathcal{V}_B^e = \{u_1, ..., v_m\}$: A finite set of environment Boolean variables.*
- *$\mathcal{V}_B^s = \{v_1, ..., v_n\}$: A finite set of system Boolean variables.*
- *θ_B : An assertion over $\mathcal{V}_B$ characterizing the initial states of $\mathcal{B}$.*
- *ρ_B : An assertion over $\mathcal{V}_B \cup \mathcal{V}_B'$ characterizing the transition relation of $\mathcal{B}$.*

Note that we do not require a base system $\mathcal{B}$ to be deterministic (although our generated controller should be deterministic and complete with respect to $\mathcal{V}_B^e$). In cases where the base system represents a 'real' concrete implementation, it would indeed be deterministic. Supporting non-deterministic base systems is useful because it enables the use of abstractions.

The concrete syntax to describe a base system is taken from the input format of SMV [1]. As an example, the SMV for the base system of our running example, which models a service for students exams, is shown in List. 1.

```
MODULE ExamService
  VARENV -- envrinoment variables
    evalExam : {pass, fail};
    newStudent : boolean;
  VAR    -- system variables
    state : {wait, welcome, inExam, diploma,
             failed, exit};
  INIT
    state=wait;
  TRANS
    ((state=wait) -> ( (next(state)=wait) |
       (newStudent & next(state)=welcome) )) &
    ((state=welcome) -> (next(state)=inExam)) &
    ((state=inExam)
      -> ( (next(state)=diploma &
            next(evalExam)=pass   ) |
           (next(state)=failed  &
            next(evalExam)=fail)  )    ) &
    ((state=diploma) -> (next(state)=exit)) &
    ((state=failed) -> (next(state)=exit)) &
    ((state=exit) -> (next(state)=wait));
```

Listing 1: The code for the ExamService base system

Roughly, when a new student comes, the service may leave the waiting state, show a welcome screen, and start the exam. The student may fail or pass the exam, after which the service moves to the exit state and back to waiting for a new student. The environment controls the input variables `evalExam` and `newStudent`. The system itself has another variable, `state`, specifying its current state. The transition system of $\mathcal{B}$ is defined in the TRANS section of the SMV file, using a conjunction of symbolically defined transitions. For example, lines 14-18 specify that whenever state = inExam, the next state may be state = diploma or state = failed, based on the value of the environment controlled input variable evalExam. Note that the base system is not deterministic: when it is waiting and a new student comes, it can either stay in waiting state or move to the welcome state. This forms an abstraction for a set of states whose details are not yet specified.

Fig. 1 shows a state diagram for the base system, projected onto the state variable.

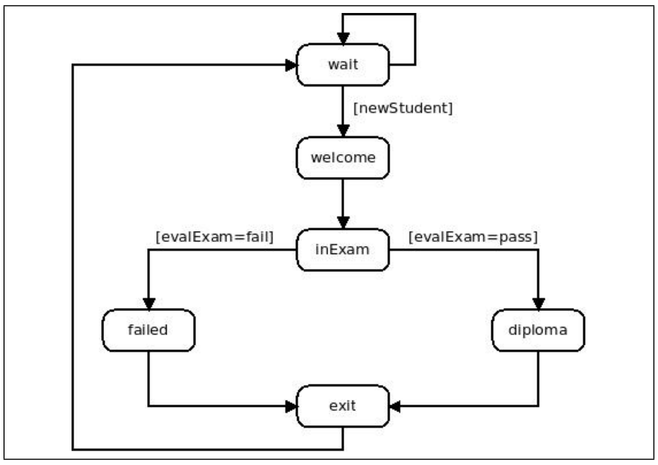

Figure 1: A state diagram for the ExamService base system (semi-formal, not all variables shown)

3.2 LTL Aspects

DEFINITION 3 (LTL ASPECT).
An LTL aspect is a structure $A = \langle \mathcal{V}_A = \mathcal{V}^e_A \cup \mathcal{V}^s_A, \theta_A, \rho_A, \mathcal{L}^s_A \rangle$ consisting of the following components:

- $\mathcal{V}^e_A$ *: A finite set of Boolean variables. $\mathcal{V}^e_A$ consists of environment variables defined at the base and are used in the aspect, denoted by $\mathcal{V}^{e_{ext}}_A$, and of new environment variables introduced by the aspect, denoted by $\mathcal{V}^{e_{new}}_A$.*
- $\mathcal{V}^s_A$ *: A finite set of Boolean variables. $\mathcal{V}^s_A$ consists of system variables defined at the base and are used in the aspect, denoted by $\mathcal{V}^{s_{ext}}_A$, and of new system variables introduced by the aspect, denoted by $\mathcal{V}^{s_{new}}_A$.*
- θ_A *: An assertion over $\mathcal{V}_A$ characterizing initial values added by A.*
- ρ_A *: An assertion over $\mathcal{V}_A \cup \mathcal{V}'_A$ characterizing transitions added by A.*
- $\mathcal{L}^s_A$ *: The aspect's LTL specification given as a formula in the form of Equ. 1 over the variables in $\mathcal{V}_A$.*

Any or all the components may be empty. If $\mathcal{L}^s_A$ is not specified it is considered to be true*.*

The concrete syntax to describe an LTL aspect is an SMV-like syntax defined as follows. A new keyword ASPECT replaces the root keyword MODULE. The environment variables $\mathcal{V}^e_A$ are defined following the keyword VARENV, and each variable is prefixed by an identifing keyword ext or new. The system's variables $\mathcal{V}^s_A$ are defined following the keyword VAR, and each variable is prefixed by an identifing keyword ext or new. The aspect's new transitions ρ_A are defined following the keyword TRANS, and the LTL specification $\mathcal{L}^s_A$ is defined following the keyword LTLSPEC.

To continue our example, we show two LTL aspects. The first LTL aspect, called Tuition, handles a security concern: it prevents new students who have not paid their tuition from taking an exam and instead redirects them to the exit state where it shows them a relevant message. The code for this aspect is shown in List. 2.

```
ASPECT Tuition
  VARENV -- envrionment variables
    new tuition : boolean;
  VAR    -- system variables
    ext state : {wait, welcome, exit};
    new showTuition : boolean;
  TRANS
    -- adding a transition from wait directly
    -- to exit if the tuition was not paid.
    ( state=wait & next(state)=exit &
      !tuition & next(showTuition) );
  LTLSPEC
    -- there is no transition from wait to
    -- welcome if the tuition was not paid.
    [] ( !(state=wait & !tuition &
           next(state)=welcome) );
```

Listing 2: The Tuition LTL aspect

The aspect declares the base variables that are used in it (in this case, state) and introduces two new variables, tuition for the environment and showTuition for the system. In the TRANS section, the aspect adds a transition from wait to exit with the special showTuition, for the case where the tuition was not paid. The aspect's LTLSPEC section specifies a 'safety' formula: there will never be a transition from wait to welcome if tuition is not paid.

The Tuition LTL aspect can be categorized as a weakly invasive aspect [16]: it restricts the behavior of the base system by removing transitions, and it adds transitions, but only between states that were reachable in the original base system. The effect of weaving it with the base system is illustrated (semi-formally) in Fig. 2.

A second example LTL aspect, called Availability, handles the service's availability concern: it specifies that whenever a new student is present when the system is waiting, eventually the system will welcome the student. The code for this aspect is shown in List. 3. The aspect's LTLSPEC section specifies a response formula, stating that globally, whenever a new student is presented and the system is in wait state, eventually the system will visit the welcome state.

The Availability LTL aspect neither adds new transitions nor explicitly restricts other transitions. However, its LTL specification may (implicitly) lead the system to choose some transitions over others.

DEFINITION 4 (ASPECTLTL SPECIFICATION).
An AspectLTL specification is a structure $\mathcal{S} = \langle \mathcal{B}, \mathcal{A} \rangle$ where $\mathcal{B}$ is a base system and $\mathcal{A} = \{A_1, A_2, .., A_k\}$ is a set of LTL aspects.

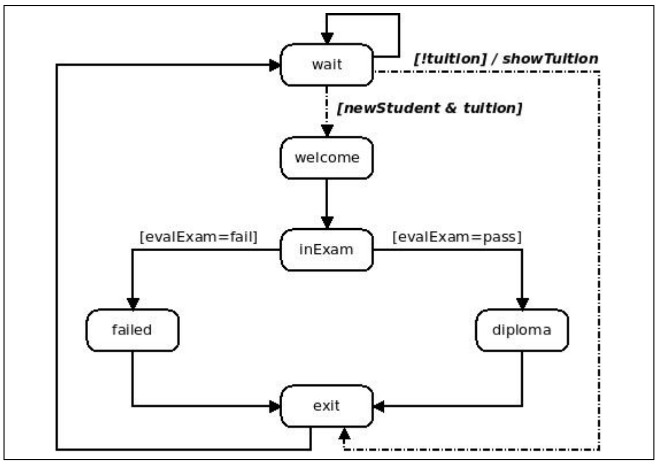

Figure 2: The `ExamService` base and the `Tuition` LTL aspect (semi-formal, not all variables shown)

```
ASPECT Availability
  VARENV -- envrionment variables
    ext newStudent : boolean;
  VAR    -- system variables
    ext state : {wait, welcome};
  LTLSPEC
    -- if the environment sends students,
    -- eventually the system accepts them and
    -- move to welcome state.
    [] ( (state=wait & newStudent)
      -> <> (state=welcome));
```

Listing 3: The `Availability` LTL aspect

We omit obvious syntactic constraints, type checking, and name space issues, e.g., that for all $A_i \in \mathcal{A}$, the external variables of $\mathcal{V}_A^s$ are indeed defined by the base system, i.e., that $\mathcal{V}_A^{eext} \subseteq \mathcal{V}_B$, and that the domains of these variables, as defined in the aspects, are subdomains of the variables domains as defined in the base.

3.3 Semantics

Definition 5 (AspectLTL implementation). *A discrete system $\mathcal{C} = \langle \mathcal{V}_C, \theta_C, \rho_C \rangle$ is an implementation of an AspectLTL specification $\mathcal{S} = \langle \mathcal{B}, \mathcal{A} \rangle$ iff the following hold:*

- $\mathcal{V}_C = \mathcal{V}_B \cup \bigcup_{A \in \mathcal{A}} \mathcal{V}_A$
- *θ_C is an initial state satisfying $\theta_B \vee \bigvee_{A \in \mathcal{A}} \theta_A$*
- *ρ_C is a subset of the transition relation satisfying $\rho_B \vee \bigvee_{A \in \mathcal{A}} \rho_A$*
- *$\mathcal{C}$ is deterministic with respect to $\mathcal{V}_B^e \cup \bigcup_{A \in \mathcal{A}} \mathcal{V}_A^e$*
- *Each run of $\mathcal{C}$ satisfies $\bigwedge_{A \in \mathcal{A}} \mathcal{L}_A^s$.*

Note that a specification defines no order between its aspects and indeed, the semantics of AspectLTL defined above is agnostic to aspect order.

Definition 6 (AspectLTL realizability). *An AspectLTL specification is realizable iff it has an implementation.*

4. WEAVING LTL ASPECTS

Given an AspectLTL specification, the goal of the weaving is to produce an implementation (if one exists). LTL aspects weaving consists of three steps: composition of game structure to model the AspectLTL implementation, checking realizability to determine whether there exists an AspectLTL implementation, and strategy extraction to produce an AspectLTL implementation (synthesis).

4.1 Composition of Aspect Game Structure

The input for the aspect game construction is an AspectLTL specification. Given an AspectLTL specification $\mathcal{S} = \langle \mathcal{B}, \mathcal{A} \rangle$, the composition of $\mathcal{S}$ is the game structure $G_B^{\mathcal{A}}$, made of the following components:

- $\mathcal{V} = \mathcal{V}_B \cup \bigcup_{A \in \mathcal{A}} \mathcal{V}_A$, i.e., all variables declared by the base system and all the aspects.
- $\mathcal{X} = \mathcal{V}_B^e \cup \bigcup_{A \in \mathcal{A}} \mathcal{V}_A^e$, i.e., all environment (input) variables declared by the base and all the aspects.
- $\mathcal{Y} = \mathcal{V}_B^s \cup \bigcup_{A \in \mathcal{A}} \mathcal{V}_A^s$, i.e., all system (output) variables declared by the base and all the aspects.
- $\theta_e = \mathsf{true}$.
- $\theta_s = (\theta_B \vee \bigvee_{A \in \mathcal{A}} \theta_A) \wedge \bigwedge_{A \in \mathcal{A}} \mathcal{L}_{A,i}^s$, where $\mathcal{L}_{A,i}^s$ is a conjunction of all i parts of $\mathcal{L}_A^s$ (part (i) in Defn. 1).
- $\rho_e = \mathsf{true}$.
- $\rho_s = (\rho_B \vee \bigvee_{A \in \mathcal{A}} \rho_A) \wedge \bigwedge_{A \in \mathcal{A}} \mathcal{L}_{A,t}^s$, where $\mathcal{L}_{A,t}^s$ is a conjunction of all t parts of $\mathcal{L}_A^s$ (part (ii) in Defn. 1).
- $\varphi = \bigwedge_{A \in \mathcal{A}} \mathcal{L}_{A,g}^s$, where $\mathcal{L}_{A,g}^s$ is a conjunction of all g parts of $\mathcal{L}_A^s$ (part (iii) in Defn. 1).

The game structure composition is implemented using standard BDD-based operations. Its complexity, i.e., the number of BDD operations, is linear to the number of input aspects.

4.2 Checking Realizability

Given an aspect game structure, checking realizability amounts to solving the game, that is, to checking who is winning, the system or the environment. Technically, this is done using a μ-calculus formulation that characterizes the set of winning states for the system player. It is computed using a symbolic BDD-based polynomial (quadratic) algorithm, that is an implementation of the nested fixpoint in the μ-calculus formulation (see [25]).

In our example, it is easy to see that the specification $\mathcal{S} = \langle \texttt{ExamService}, \{\texttt{Tuition}\} \rangle$ is realizable, since all states of the system player defined in the aspect game structure and reachable from the initial states are winning for the system: there is no additional liveness requirement to take care of, so as long as the system does not get stuck, it will win.

The specification $\mathcal{S} = \langle \texttt{ExamService}, \{\texttt{Availability}\} \rangle$ is realizable too. Again all reachable states defined in the aspect game structure are wining for the system player, although it needs to be careful, to indeed move to the `welcome` state infinitely often (and not stay in the `wait` state forever), so it will satisfy the response property of `Availability`.

Interestingly however, the AspectLTL specification $\mathcal{S} = \langle \texttt{ExamService}, \{\texttt{Tuition}, \texttt{Availability}\} \rangle$ is *not* realizable, since the initial `wait` state is not winning for the system: an environment that sends infinitely many students such that only finite number of them have paid their tuition, will force the system not to visit the `welcome` state infinitely often, and thus to violate its specification.

To 'fix' that, we suggest another, more general, version for the `Availability` aspect, `AvailabilityNotWait`, where we fix the `LTLSPEC` to read: $\Box((\texttt{state} = \texttt{wait}\ \&\ \texttt{newStudent}) \rightarrow (\texttt{state}\ != \texttt{wait}))$, forcing the system to leave the `wait` state but not necessarily visit the `welcome` state. With this new version of the aspect we have that the specification $\mathcal{S} = \langle \texttt{ExamService}, \{\texttt{Tuition}, \texttt{AvailabilityNotWait}\}\rangle$ is realizable.

If the specification is realizable, we extract one winning strategy from it and present it as an executable automaton (see next). Otherwise, if the AspectLTL specification is not realizable, further analysis can be done to investigate the reason for failure (see Sec. 7).

4.3 Strategy Extraction (Synthesis)

In many cases, there may be more than one winning strategy for the system. For example, a winning strategy for the AspectLTL specification $\mathcal{S} = \langle \texttt{ExamService}, \{\texttt{Availability}\}\rangle$ may advance from `wait` to `welcome` only after taking some self transitions in the `wait` state: as long as it eventually moves to the `welcome` state the strategy will satisfy the specification.

We thus arbitrarily extract one winning strategy from the game solution. The extracted strategy induces a deterministic and complete discrete system (a controller) whose all runs are guaranteed to meet the specification, hence, it is indeed an implementation of the AspectLTL specification, as required.

Given the controller's representation, we can create an interactive program that simulates it: the program waits for assignments to input variables from the user (or from another program), advances its internal representation of the automaton accordingly, then again waits for the next assignment to input variables, and so on and so forth, ad infinitum. Indeed, our prototype implementation generates a correct-by-construction Java code for such an interactive program, so that the engineer can actually run it (see Sec. 8).

Theorem 1 (Correctness). *Given an AspectLTL specification $\mathcal{S}$, the composition and synthesis process described above yields an implementation of $\mathcal{S}$ iff $\mathcal{S}$ is realizable.*

The proof of this theorem follows our construction of the aspect game structure and requires technical details appearing in [25]. We omit it from this version of the paper.

5. ADDITIONAL EXAMPLES

In addition to the two LTL aspects presented in the previous sections, we now provide several other aspects that demonstrate various features of AspectLTL. The aspects relate to the `ExamService` base system of List. 1 discussed in the previous sections.

5.1 A logging aspect

The following example demonstrates the use of AspectLTL for logging, a 'classical' use of aspects.

Consider a very simple LTL aspect named `FailuresLogging`, intended to work as a logger for the `ExamService` base system. The aspect is responsible for logging the details of students that fail the exam, when they exit the system. The AspectLTL code for `FailuresLogging` appears in List. 4.

```
ASPECT FailuresLogging
  VAR     -- system variables
    ext state : {failed, exit};
    new logFail : boolean;
  LTLSPEC
    -- logFail iff the previous state was
    -- failed and the current state is exit
    !logFail &
    [] (((_)(state=failed) & state=exit)
      <-> logFail);
```

Listing 4: The `FailuresLogging` LTL aspect

Logging is modeled using a new system variable named `logFail`. The aspect uses a past formula (with the *previous* operator $\ominus\varphi$ written `(_)`), because the failure information is available in the `failed` state but needs to be logged in the `exit` state.

Note that `FailuresLogging` is very simple: it neither restricts the behavior of the base system in any way nor adds new states or transitions. It can be categorized as a spectative aspect [16].

5.2 Adding exceptional behavior and choosing between futures

The following example demonstrates the power of AspectLTL correct-by-construction weaving to resolve possible non-trivial aspect interferences.

The LTL aspect `AllowQuitting` shown in List. 5 specifies the addition of exceptional behavior: at any time during the run, the student who uses the exam service may ask the system to cancel. Moreover, the aspect specifies that when the user asks to cancel, the system will set `quit` to `true`. A new transition expression is added to specify that when `quit` is `true`, the system can move to the `wait` state.

```
ASPECT AllowQuitting
  VARENV -- envrionment variables
    new cancel: boolean;
  VAR     -- system variables
    ext state : {wait};
    new quit: boolean;
  TRANS
    -- when quit is set, adding transition from
    -- anywhere to wait
    (quit & next(state)=wait);
  LTLSPEC
    -- if moving to wait then resetting quit
    [] ((next(state)=wait) -> !next(quit)) &
    -- otherwise,
    [] (next(state)!=wait
        -- if cancel request, then quit
      ->(( next(cancel) -> next(quit)) &
        -- if no cancel request, then idle
        (!next(cancel) -> next(quit)=quit))) &
    -- if quit is set, then eventually moving
    -- to wait
    [] (quit -> <> (state=wait));
```

Listing 5: The `AllowQuitting` LTL aspect

Note that the addition of the behavior, from `quit` to the `wait` state, in the `TRANS` section of the aspect, is done symbolically: explicit individual states are not mentioned in the transition definition; one short formula covers a set of new concrete transitions. Note also that the augmented system is now non-deterministic: if the user asks to cancel, the system sets the `quit` variable, but then it may move to the `wait` state next or sometime in the future (as expressed in the `LTLSPEC` section of the aspect). An implementation may choose not to move to the `wait` state immediately, and it will be correct as long as it gets there eventually.

One ‘problem’ with this aspect, from a requirements point of view, is that the system may allow the student to cancel after she has seen the exam but without having her grade recorded. Obviously, this is not acceptable. The next aspect, named `ExamProtection` (shown in List. 6), ensures that this will indeed not happen: it specifies that when the system is in the `inExam` state, eventually it will be in the `exit` state and it will not quit to the `wait` state since it has started the exam in the `welcome` state.

```
ASPECT ExamProtection
  VAR     -- system variables
    ext state : {wait, welcome, inExam, exit};
  LTLSPEC
    -- if we are in exam, then eventually we
    -- will exit, and we won't be in wait since
    -- that exam.
    [] (state=inExam -> <> (state=exit &
            (state!=wait Since state=welcome)));
```

Listing 6: The `ExamProtection` LTL aspect

Consider the aspect game structure for the specification $\mathcal{S} = \langle \texttt{ExamService}, \{\texttt{AllowQuitting}, \texttt{ExamProtection}\}\rangle$, when the system state is `inExam` and the environment (user) has set `cancel` to `true`. In this game position, the `quit` variable must be `true`, so the system player has the option whether or not to move to the `wait` state. However, while the transition to the `wait` state is indeed possible, if the system player chooses it, the resulting game state will be winning for the environment player: from this game state, the environment player can force the system to violate the specification, by sending a new student and waiting for the system to eventually reach the `exit` state. The specification of `ExamProtection` will eventually be violated because the system will visit the `wait` state on the way from `inExam` to `exit`.

Indeed, when computing the solution to the aspect game structure, the synthesis algorithm will find out that this state is winning for the environment and thus will not include the transition to it in the system's winning strategy. Note that this kind of reasoning relays on the μ-calculus nested fixpoint algorithm's ability to ‘look ahead’ and ‘backtrack’ when marking game states as winning for the system player or the environment player.

This example shows the power of AspectLTL correct-by-construction weaving to resolve possible non-trivial aspect interferences.

5.3 Adding a bounded loop

Finally, an example for adding a behavior in the form of a bounded loop.

The LTL aspect `ExamCounter` shown in List. 7 defines a ‘retry’ feature: a student who fails the exam may repeat it up to 3 times. The aspect defines an environment controlled variable `tryAgain` (the user may choose whether to try again or not), and a system controlled variable `counter`. In addition, the aspect adds a transition from the `failed` state back to the `inExam` state, which is possible only when the counter is less than 3 and the user has asked to retry. Finally, the aspect specifies the conditions for initializing the counter, incrementing it, and resetting it.

```
ASPECT ExamCounter
  VARENV -- envrionment variables
    new tryAgain : boolean;
  VAR    -- system variables
    ext state : {welcome, inExam, failed};
    new counter : {0, 1, 2, 3}
  TRANS
    -- add transition from failed back to exam
    (  state=failed & next(state)=inExam &
       (counter<3) & tryAgain );
  LTLSPEC
    (counter=0) &
    -- if taking new transition, then counter++
    [] ( (state=failed & counter<3 & tryAgain)
      -> ( next(state)=inExam &
           next(counter)=(counter+1) ) ) &
    -- if in welcome, then reseting conuter
    [] ( state=welcome -> next(counter)=0 ) &
    -- otherwise, leaving the counter as is
    [] (!( state=welcome |
           (state=failed & next(state)=inExam))
      -> next(counter)=counter );
```

Listing 7: The `ExamCounter` LTL aspect

Fig. 3 illustrates the effect of weaving the `ExamCounter` aspect to the base system.

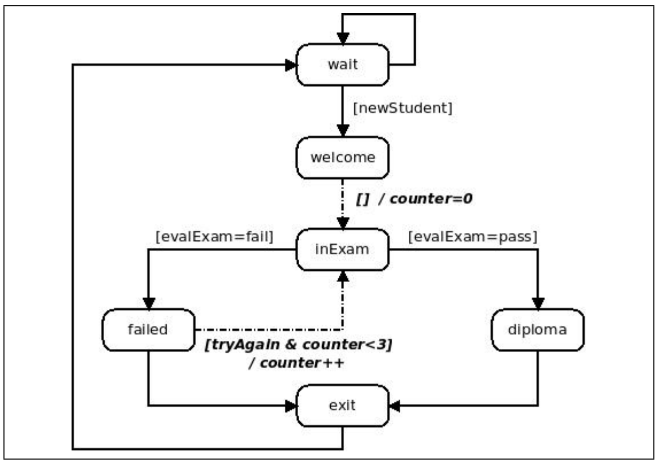

Figure 3: The effect of weaving the `ExamCounter` aspect to the `ExamService` base system (semi-formal, not all variables and transitions shown)

6. LANGUAGE EXTENSIONS

In this section we discuss several extensions to the basic version of AspectLTL presented in the previous sections.

6.1 Environment assumptions

An important, natural extension of the basic version of AspectLTL is the addition of environment assumptions. Rather than requiring the system specification to hold in *all* environments, it is common to restrict the specification of open systems with assumptions on environment behavior. Roughly, this results in an assume/guarantee style specification: if the environment adheres to its specification (the assumptions), the system must adhere to its specification (the guarantees).

The algorithms presented in [25] support this assume/guarantee style formula (the GR(1) fragment). In the reduction given above we used these algorithms but just fixed the assumptions to be `true`, meaning no restrictions were specified on the environment. Thus, it is indeed possible to extend AspectLTL with support for environment assumptions, based on existing synthesis algorithms and while staying within the polynomial complexity[3].

We now sketch the adaptation. Syntactically, the addition of support for environment assumptions is implemented by the addition of a new keyword `LTLSPECENV`, where a formula of the form of Equ. 1 is placed, specifying what is assumed to hold for the environment. In the weaving process, the aspect game structure composition defined in Subsection 4.1 needs to change. In particular, θ_e and ρ_e is not fixed to `true`, and the winning condition formula φ includes expressions from part (iii) in Defn. 1 of the formulae defined inside the new `LTLSPECENV` sections.

Supporting assume/guarantee specification enables the use of richer and more realistic specifications. For lack of space in this proceedings we omit the formalization and concrete examples of this extension from this version of the paper.

6.2 Changing variable domains

The basic version of AspectLTL presented in this paper allows aspects to access system and environment variables (referring to the entire domain or to a locally defined subdomain of each variable), and to introduce new system and environment variables. However, it does not allow an aspect to change the domain of existing system or environment variables.

The effect of changing the domain of a variable can be simulated by introducing new variables and creating dependencies between the new variables and the existing ones. Thus, formally, adding this feature to AspectLTL does not increase the language's expressive power. Still, in some cases, it may be more elegant to directly change the domain of a variable: other aspects, which may be co-weaved with the augmented base system, will 'see' the new domain rather than the original one. They will be able to use the new domain without being 'aware' that it was redefined by another aspect.

Changing the domain of an existing variable may be viewed as a variant of *static crosscutting*, adapted to AspectLTL. It needs to be used with care, but it does create an opportunity for cleaner end results.

6.3 Visibility and scope

It is useful to extend AspectLTL with notions of variable visibility and scope.

For example, allow the base system to distinguish between variables that are *public* and can be accessed by aspects, and variables that are *private* and cannot be accessed by aspects. The definition of AspectLTL in Sec. 3 assumes all variables are public.

Moreover, it is useful to define a notion of scope, whereby the access to base or environment variables is done using imported name spaces or explicitly with fully qualified names (e.g., `base.state`). The definition of AspectLTL in Sec. 3 assumes a single, global name space.

Such notions of visibility and scope allow structural modularity, information hiding, and aspect reuse, offering ways to better manage large AspectLTL specifications. We omit the formalization of these features from this version of the paper.

6.4 Join point model

An aspect language has a *join point model* (JPM), which defines the points where an aspect may interfere with a base, how these points may be specified, and how the additional aspect behavior is defined.

In AspectLTL we have defined a very general and permissive JPM. We allow new transitions to be added at any state of the base system: all states are possible *join points*, *pointcuts* are not specified explicitly, and *advice* is divided between the explicit transitions added in the `TRANS` section and the more implicit restrictions defined in the `LTLSPEC` section.

Note the unique characteristics of AspectLTL in this sense. The "advice" of LTL aspects has not only local and specific effect to explicitly defined points along the execution, but also a global, temporal and general effect on ongoing, infinite executions.

Nevertheless, one may consider a more restrictive JPM, which would force the engineer to explicitly define the states in the base system where an aspect's new transitions should 'start from' (using *pointcuts*) and the states where the aspect transitions end and return to the base (using *return points*) (such a restricted JPM was used in [12] in the context of aspects model-checking). Note that these pointcuts and return points may still be defined symbolically, that is, using expressions that define a set of states, not a single state.

We see no major difficulty in introducing such a restricted JPM to AspectLTL. Having it may be useful when reducing other (aspect) languages into AspectLTL (see 7.2).

7. DISCUSSION OF ADVANCED TOPICS

We now discuss several advanced topics and future work directions related to AspectLTL and its possible applications.

7.1 Expressive power and complexity

The expressive power of aspects, to specify behavior and to manipulate a base program, is an important issue in the design of aspect languages. We have shown how AspectLTL enables the specification and enforcement of liveness and safety concerns, as well as the addition of new behaviors, covering spectative, regulate, and invasive aspects [16].

Formally, the basic version of AspectLTL presented in this paper supports part of the GR(1) fragment of LTL, limited to system specifications in the form of Equ.1. The GR(1) fragment supports specifications of the form $\varphi \rightarrow \psi$ where φ and ψ are both in the form of Equ.1 (that is, the addition of environment assumptions, see subsection 6.1). A conjunction of GR(1) specifications, called a GR(K) specification, is at the

[3]We choose not to include environment assumptions in the basic AspectLTL version presented above, so as to keep it simpler and more intuitive to understand. The full extent of the GR(1) fragment may be viewed as carrying with it a somewhat counter-intuitive interpretation with respect to natural programming constructs. In order to consider it, one would have to rigorously define a gray area in which programming is still natural while allowing assumptions on the environment.

top of the hierarchical classification of LTL properties presented in chapter 4 of [22]. That is, every LTL specification can be transformed into GR(K) (albeit with non-elementary complexity).

The general case for synthesis from LTL specifications is 2EXPTIME-complete [28] (however, solving a GR(K) game is exponential in k [24]). As shown in [25], synthesis from the GR(1) fragment is polynomial. As AspectLTL weaving is limited to the GR(1) fragment (even with the addition of environment assumpotions), it uses efficient BDD-based symbolic algorithms. It is linear to the number of aspects in the specification and polynomial (quadratic) to the state space.

7.2 Reducing other languages into AspectLTL

One may consider using AspectLTL to produce correct-by-construction controllers from programs written in other languages. Given the high expressive power of AspectLTL, we believe it would be possible to define a semantics preserving translation from, e.g., a variant of AspectJ [18] annotated with aspect's pre/post conditions, into AspectLTL, and use AspectLTL weaving to reason about the alternative weaving orders or to directly produce the final executable artifact.

Such a translation requires the extraction of finite state machine models from the base java program and from the AspectJ aspects (which may be done automatically using a tool like Bandera [7]). It may also require the restriction of AspectLTL join point model to pointcuts and advice similar to the ones of AspectJ (see subsection 6.4).

As an example, AspectLTL weaving may be used in this case to find a non-conflicting advice execution order in case of ambiguities that result in possible interferences, or to prove that any weaving order will not satisfy the aspects specifications. Moreover, it may be possible to use LTL aspects directly on the finite state machine model extracted from the base Java program, regardless of the use of AspectJ.

7.3 Debugging unrealizable specifications

The examples we have shown in previous sections demonstrated the power of AspectLTL's weaving process to identify possible interferences or conflicts between LTL aspects and solve them, if possible, so as to generate a controller that implements the specification. In particular, see the example in Sec. 5.2.

When an AspectLTL specification is not realizable, a controller is not generated. In this case, one may be interested in debugging the specification, in order to identify and understand the reason for unrealizability.

One possible way to debug an unrealizable specification may be based on counterstrategies [20]. By reversing the roles of the system and the environment in the synthesis game, we are able to generate winning strategies for the environment. Following these strategies shows exactly how *any* generated system can be forced by an (adverse) environment to violate the system specifications.

Recalling the example LTL aspects presented in Sec. 3 and discussed in subsection 4.2, we explained that the AspectLTL specification $\mathcal{S} = \langle$`ExamService`, $\{$`Tuition`, `Availability`$\}\rangle$ is *not* realizable: an environment that sends infinitely many students such that only finite number of them have paid their tuition, will force the system not to visit the `welcome` state infinitely often, and thus to violate its specification. The reversing roles process described above is able to synthesize this environment strategy.

A related technique for the debugging of unrealizable AspectLTL specifications is the computation of an unrealizable core [6]. An unrealizable core is a minimal subset of the specification that is unrealizable (note that there could be several such cores). Presenting an unrealizable core to the user may be viewed as a form of program slicing, allowing to localize the cause of bugs. The larger the specification, in terms of the number of LTL aspects involved, the more critical the computation of the core becomes.

7.4 Traceability

Traceability, the ability to trace a systems requirements from the specification to the implementation, is an important issue in software engineering. In particular, the use of an aspect language poses opportunities and challenges for traceability.

In the context of AspectLTL, traceability would mean the ability to identify, for each transition of the generated controller, whether it is defined by the base system or by one of the aspects, and, in case of non-deterministic choice in the initial or intermediate outcomes, which aspect specification led the strategy to choose one of the possible transitions over the others.

Recalling our examples again, in the AspectLTL specification $\mathcal{S} = \langle$`ExamService`, $\{$`Tuition`, `AvailabilityNotWait`$\}\rangle$, the transition from `wait` to `exit` should be traced back to the `Tution` aspect, and the transition from `wait` to `welcome` should be attributed both to the base system and to the `AvailabilityNotWait` LTL aspect.

We believe that such traceability features can be implemented, automatically, by programmatically adding supportive variables (during the composition of the aspect game structure), which are not necessary for the controller generation but will 'label' its transitions with the required information. We leave the formalization, automation, and user-interface presentation of this feature to future work.

8. IMPLEMENTATION

AspectLTL is supported by a prototype Eclipse plug-in, which we have developed on top of JTLV [29], a framework for the development of verification algorithms, using BDD-based symbolic mechanisms. The default underlying BDD package used is CUDD [31]. We used this prototype implementation to define several AspectLTL specifications (including the examples we presented in this paper), to weave them, and to run the generated implementations.

The plug-in implements Eclipse editors for AspectLTL base and aspect formats, including syntax coloring etc. It allows the engineer to check the realizability of an AspectLTL specification, to synthesize, and to output the synthesis result. The result is presented to the engineer in either a plain text format, or in a generated code, consisting of an interactive Java program. The generated program simulates the system's winning strategy: it asks the user for values for the input variables, advances its internal representation of the automaton accordingly, then again waits for the next assignment to input variables, and so on and so forth, ad infinitum.

Thanks to the use of BDDs and the algorithms of [25], the complexity of AspectLTL composition and synthesis is linear to the number of aspects and polynomial (quadratic) to the state space. Hence, performance is not expected to be a major problem, at least up to medium scale designs. For example, the composition and synthesis of a specification that consists of the example base system `ExamService` and all 6 example LTL aspects we have presented in this paper, takes about 3.5 seconds on a standard laptop machine. Out of these, parsing, BDD factory setup, and compositions take approximately 0.1 seconds, realizability check takes approximately 0.5 seconds, and the remaining approximately 3 seconds are used for the strategy extraction and Java code generation. Further performance experiments with larger case studies are left for future work.

9. RELATED WORK

We now discuss related studies in the area of specifying and modeling aspects using LTL or state machines and in the area of composition and program synthesis.

Goldman and Katz [12] model an aspect using an SMV-like format and describe a modular verification technique, based on model checking, allowing one to prove the correctness of an aspect once, for all base systems satisfying the aspect's assumption about the systems it may be woven to. The aspect advice is described using a state machine. Pointcuts and return states are described using LTL expressions (however, pointcuts can only use current-state expressions, past LTL syntax is not supported). Their work is extended in [15] to support strongly-invasive aspects.

Krishnamurthi et al. [21] model a program's control-flow graph as a state machine, model an advice as another state machine, and specify pointcuts using a (restricted form of) regular expressions. They show how CTL properties that were proved to hold for the base program can be proved to hold for the augmented program in a modular way, without analyzing the entire system every time a developer changes an advice.

The above mentioned works use the temporal specification or state machine of the aspect as a means towards a model-checking based proof of aspects correctness. Other works translate LTL properties into corresponding monitors written using aspect code (e.g., in AspectJ), as a means for LTL runtime verification (see, e.g., [13, 32]).

In contrast to the above lines of work, we use an LTL characterization of aspects as an input for a composition and synthesis process. Rather than static or runtime verification, our primary goal is to produce a correct-by-construction executable system. Moreover, AspectLTL features a more general and permissive joint point model, not limited apriori to AspectJ-like pointcuts and advice. Our method solves the possible conflicts or interferences between the specified aspects, if indeed a solution to these conflicts exists. If a solution does not exist, a static debugging mechanism may be used to uncover the reasons for the conflicts (see Sec. 7.3). We are not aware of any other work that uses an LTL charaterization of aspects, or an LTL-based aspect language like AspectLTL, for the purpose of correct-by-construction aspect weaving.

It is important to distinguish AspectLTL synthesis from other forms of composition and program synthesis that have attracted research attention in recent years. Correct composition of features (see, e.g., [8, 14, 33]), for example, is typically discussed at the level of safe type checking, and not at the level of the actual semantics of the features involved: features can be composed if the resulting program is type-safe and compiles, not if its semantics (in terms of input/output or event sequences) indeed complies with each of the features' specifications.

Dependencies between features are typically expressed using logical constraints at the level of the feature model (e.g., "if feature A is included, feature B must not be included"). Feature models may be the subject of formal analysis (see, e.g., [23, 34]). In contrast, in AspectLTL, in addition to type safety, correct composition and synthesis depends on the actual semantics of the LTL aspects specifications. Moreover, dependencies between LTL aspects are not provided by the designer. Instead, they are automatically and implicitly deduced from the actual aspect code during the synthesis process. Thus, indeed, AspectLTL synthesis is very different and complementary to these approaches.

10. CONCLUSION AND FUTURE WORK

We presented AspectLTL, a temporal-logic based language for the specification and implementation of crosscutting concerns. AspectLTL enables the modular declarative specification of expressive concerns, covering the addition of new behaviors, as well as the specification of safety and liveness properties. Moreover, given an AspectLTL specification, consisting of a base system and a set of aspects, AspectLTL comes with a composition and synthesis-based weaving process, whose output is a correct-by-construction executable artifact. We formally defined the syntax and semantics of AspectLTL, presented several example LTL aspects, and described a prototype implementation that supports the new language.

AspectLTL is an *aspect language*, as it indeed provides quantification and obliviousness [11]. The use of the symbolic interpretation provides quantification: unitary and separate statements have effect in many non-local places in the system. Obliviousness is supported, as the base system does not have to be specifically prepared to receive the enhancements defined by the LTL aspects.

Several future work directions were discussed already in Sec. 7, among them: implementing a debugger based on counterstrategies for better interference analysis, and defining reductions from other languages into AspectLTL.

Additional future work directions follow.

First, to make the writing of AspectLTL aspects more human-firendly and the resulting text more human-readable, we plan to define a set of specification patterns that can be used inside LTL aspects specifications as macros (syntactic sugars) (as in, e.g., [9]). For example, supporting an `idle` keyword, $idle(v_1, v_2, \ldots, v_k)$ as a syntactic sugar for $\&_{i=1}^{k} next(v_i) = v_i$, or a `loop` keyword to iterate at a given state up to a finite number of times. Note that we already support several such patterns, e.g., the response pattern $\Box(\varphi \rightarrow \Diamond \psi)$, and that the implementation of such patterns in the context of our synthesis algorithm requires a transformation to the format of Equ. 1, hence it is not straightforward and may require the automatic implicit addition of helper variables.

Second, we plan to investigate LTL aspects reuse. Some of the example LTL aspects we have presented are specific to the `ExamService` base system, but others are relatively generic and may be useful when weaved with many base systems. Better support for reuse may include mechanisms for defining *parameterized* LTL aspects, with more generic mappings (bindings) between aspect and base variables, and means to specify aspect's assumptions about the base systems it may be woven to (which can be verified prior to composition and synthesis).

Acknowledgments

We would like to thank Nir Piterman for comments on a draft of this paper.

11. REFERENCES

[1] SMV model checker. http://www.cs.cmu.edu/~modelcheck/smv.html.

[2] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Design Automation and Test in Europe*, pages 1188–1193, 2007.

[3] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. In *6th Int. Workshop on Compiler Optimization Meets Compiler Verification*, volume 190(4) of *ENTCS*, pages 3–16, 2007.

[4] J. Büchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[5] A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25, Upsala, 1963.

[6] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *VMCAI*, volume 4905 of *LNCS*, pages 52–67. Springer, 2008.

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE*, pages 439–448, 2000.

[8] B. Delaware, W. R. Cook, and D. S. Batory. A machine-checked model of safe composition. In *FOAL*, pages 31–35. ACM, 2009.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420. ACM, 1999.

[10] E. A. Emerson and C. L. Lei. Efficient model-checking in fragments of the propositional modal μ-calculus. In *Proc. 1st IEEE Symp. Logic in Comp. Sci.*, pages 267–278, 1986.

[11] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[12] M. Goldman and S. Katz. MAVEN: Modular aspect verification. In *TACAS*, volume 4424 of *LNCS*, pages 308–322. Springer, 2007.

[13] H. Goldsby, B. H. C. Cheng, and J. Zhang. AMOEBA-RT: Run-Time Verification of Adaptive Software. In *MoDELS Workshops*, volume 5002 of *LNCS*, pages 212–224. Springer, 2007.

[14] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *ASE*, pages 258–267. IEEE, 2008.

[15] E. Katz and S. Katz. Modular verification of strongly invasive aspects: summary. In *FOAL*, pages 7–12. ACM, 2009.

[16] S. Katz. Aspect categories and classes of temporal properties. In A. Rashid and M. Aksit, editors, *T. Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 106–134. Springer, 2006.

[17] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Inf. and Comp.*, 163:203–243, 2000.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[20] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD*, pages 152–159. IEEE, 2009.

[21] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT FSE*, pages 137–146. ACM, 2004.

[22] Z. Manna and A. Pnueli. *The temporal logic of concurrent and reactive systems: specification.* Springer, 1992.

[23] M. Mannion. Using first-order logic for product line model validation. In *SPLC*, volume 2379 of *LNCS*, pages 176–187. Springer, 2002.

[24] N. Piterman and A. Pnueli. Faster solutions of rabin and streett games. In *LICS*, pages 275–284. IEEE Computer Society, 2006.

[25] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. In *VMCAI*, volume 3855 of *LNCS*, pages 364–380. Springer, 2006.

[26] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[27] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.

[28] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, volume II, pages 746–757. IEEE, 1990.

[29] A. Pnueli, Y. Sa'ar, and L. Zuck. JTLV: A framework for developing verification algorithms. In *CAV*, 2010.

[30] M. Rabin. *Automata on Infinite Objects and Churc's Problem*, volume 13 of *Regional Conf. Series in Mathematics.* Amer. Math. Soc., 1972.

[31] F. Somenzi. CUDD: CU Decision Diagram package. http://vlsi.colorado.edu/~fabio/CUDD/, 1998.

[32] V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *5th Workshop on Runtime Verification*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 109–124. Elsevier, 2005.

[33] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe composition of product lines. In *GPCE*, pages 95–104. ACM, 2007.

[34] T. Thüm, D. S. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE*, pages 254–264. IEEE, 2009.