# Two-Way Traceability and Conflict Debugging for AspectLTL Programs *

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

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

## Abstract

Tracing program actions back to the concerns that have caused them and blaming specific code artifacts for concern interference are known challenges of AOP and related advanced modularity paradigms. In this work we address these challenges in the context of AspectLTL, a temporal-logic based language for the specification and implementation of crosscutting concerns, which has a composition and synthesis-based weaving process whose output is a correct-by-construction executable artifact. When a specification is realizable, we provide two-way traceability information that links each allowed or forbidden transition in the generated program with the aspects that have justified its presence or elimination. When a specification is unrealizable, we provide an interactive game proof that demonstrates conflicts that should be fixed. The techniques are implemented and demonstrated using running examples.

***Categories and Subject Descriptors*** D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Languages, Design

***Keywords*** Aspect-oriented programming, linear temporal logic, synthesis

* 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*AOSD'12,* March 25-30, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1092-5/12/03. . . $10.00

## 1. Introduction

Separation of concerns at the source code level supports cleaner, modular designs, but may also make the traceability and debugging of the implementations more technically challenging. Thus, tracing program actions back to the concerns that have caused them and blaming specific code artifacts for concern interference are known challenges related to AOP and other advanced modularity paradigms.

AspectLTL [17] is a temporal-logic based language for the specification and implementation of crosscutting concerns in open reactive systems – discrete event systems that maintain ongoing interaction with their environment. An AspectLTL specification is made of a base system, given as a finite-state machine specified in SMV [23] 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. The language has a composition and synthesis-based weaving process, based on GR(1) synthesis [19], whose output is a correct-by-construction executable artifact. An implementation of AspectLTL that produces a stand-alone Java controller was described in [17].

In this work we address the traceability and debugging challenges in the context of AspectLTL. When a specification is realizable, we provide two-way traceability information that links each allowed or forbidden transition in the generated program with the aspects that have justified its presence or elimination. When a specification is unrealizable, we provide an interactive game proof that demonstrates conflicts that should be fixed.

AspectLTL uses a declarative, symbolic programming style. For specifications that consist of several aspects and describe some possibly crosscutting concerns, traceability and debugging are *very different* than their counterparts in imperative programming languages. In particular, the presence or elimination of a behavior (i.e., a transition from one state to another) in the program, may be the result of the application of several, possibly overlapping, non-independent concerns. To support traceability, we use symbolic operations to check for intersections between the transitions that can or cannot be taken and the formulas defined in the LTL aspects. To support conflict debugging in unrealizable specifications we use the notion of counterstrategies [13]. By reversing the roles of the system and the environment in the synthesis game, we are able to generate a winning strategy for the environment. We use this strategy to produce a counter-implementation: an interactive program, whose runs show exactly how any generated system can be forced by an (adverse) environment to violate the specifications.

One may question the need for traceability and debugging for a language with a 'correct-by-construction' implementation: if the implementation is 'correct-by-construction', who needs debugging? Correctness, however, is relative to the specification: if the engineer writes a conflicting, unrealizable specification, no correct implementation can be generated. Instead, a conflict debugging technique should be used to prove unrealizability and point the engineer to conflicts in her specification. Thus, the 'correct-by-construction' implementation does not eliminate the need for debugging: it lifts debugging from the concrete implementation to the higher-level, declarative specification.

Further, one may question the need for traceability for realizable specifications: if the specification is realizable and the implementation is 'correct-by-construction', who needs traceability? Correctness of a realizable specification is, however, relative to the engineer's intention: when running the generated controller (the generated Java program) of a realizable specification, the engineer may experience some behavior she had not intended to be possible or miss some behavior she had intended to be performed. This means that although the specification is consistent, that is, mathematically, it is too permissive or too restrictive with regard to the engineer's intention. If the generated program does something that was not intended, the engineer can use traceability to check which part of the specification allows it. If the generated program does not do something that was intended, the engineer can use traceability to check which aspect has prevented it.

The new traceability and debugging techniques are implemented in the AspectLTL plug-in, available from our website [2], together with several running examples. We encourage the interested reader to try them out.

Finally, in a related line of research we are working on the addition of environment assumptions to AspectLTL syntax (in a dedicated LTLSPECENV section) and semantics (in the synthesis phase), thus taking full advantage of the expressive power of GR(1) [19]. In this paper, however, we chose to focus on the traceability and debugging techniques and limit most of the discussion to the AspectLTL fragment defined in [17], which does not contain environment assumptions. Nevertheless, where relevant, we mention assumptions in several places in the paper and discuss their possible effect on the traceability and debugging techniques.

Sect. 2 provides background on AspectLTL. Sect. 3 presents the running example we use. Sect. 4 defines and demonstrates traceability, Sect. 5 extends our example with additional aspects that are used in Sect. 6, which defines and demonstrates conflict debugging. Implementation and evaluation are presented in Sect. 7. Sect. 8 discusses related work and Sect. 9 concludes.

## 2. Preliminaries

### 2.1 An overview of AspectLTL

AspectLTL [17] is a language for the specification and implementation of crosscutting concerns, based on linear temporal logic (LTL) [20]. 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 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 [11], LTL aspects can specify spectative, regulative, and invasive aspects.

AspectLTL has 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 [21], essentially a game between the environment and the (augmented) base system. An algorithm based on [19] 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.

An AspectLTL specification is made of a base system and a set of LTL aspects. The base is given as a finite-state machine specified in SMV [23] format. Each of the LTL aspects is specified in a similar SMV-like format, containing a symbolic representation of the aspect's added behaviors (transitions) and a related LTL specification.

AspectLTL is supported by an Eclipse plug-in, developed on top of JTLV [22], a framework for the development of verification algorithms, using BDD-based symbolic mechanisms. The plug-in is available from our website [2], with several examples and a programmer's guide. It includes support for editing and synthesizing AspectLTL programs, that is, it uses the results of the synthesis to generate an executable artifact in the form of stand-alone controller written in Java. As part of our work in the present paper, we have implemented the traceability and debugging features we describe and have integrated them into the plug-in.

A thorough and formal account of AspectLTL appears in [17]. Here we provide only the definitions that are required in the later parts of the paper.

## 2.2 Definitions (from [17])

We use the usual LTL notations (defined in [16, 20]), $\bigcirc$ (*next*), $\diamondsuit$ (*eventually*) and $\square$ (*globally*), abbreviations of the Boolean connectives $\wedge$, $\rightarrow$ and $\leftrightarrow$, and the usual definitions for true and false.

A *discrete system* (DS) [12] 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. [1] A *state* $s$ is an interpretation of $\mathcal{V}$, i.e., $s \in \Sigma_{\mathcal{V}}$.
- $\theta$ : 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 $\theta$.
- $\rho$ : A *transition relation*. This is an assertion over the variables in $\mathcal{V} \cup \mathcal{V}'$, 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 $\sigma$ is maximal if either $\sigma$ 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 $\varphi$, denoted $\mathcal{D} \models \varphi$, if every run of $\mathcal{D}$ satisfies $\varphi$.

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*.

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 environment's choice of assignments to input variables.

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. AspectLTL is based on 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)* *$\varphi_i$ is a Boolean formula which characterizes the initial states of the implementation.*

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

*(iii)* *$\varphi_g$ is a formula of the form $\bigwedge_{i \in I} \square \diamondsuit B_i$ where each $B_i$ is a Boolean formula. $\varphi_g$ characterizes liveness requirements for the implementation.*

[19] 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 [19] is good for our needs.

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

- *$\mathcal{V}^e_B = \{u_1, ..., v_m\}$ : A finite set of environment variables.*
- *$\mathcal{V}^s_B = \{v_1, ..., v_n\}$ : A finite set of system variables.*
- *$\theta_B$ : An assertion over $\mathcal{V}_B$ characterizing the initial states of $\mathcal{B}$.*
- *$\rho_B$ : 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 $\mathcal{V}^e_B$). 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.

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:*

[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 here.

[2] also known as the temporal semantics of *just discrete systems* (JDS) [12].

- $\mathcal{V}^e_A$ *: A finite set of variables.* $\mathcal{V}^e_A$ *consists of environment variables that are defined at the base and 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 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* $\mathcal{V}^{s_{new}}_A$*.*
- $\theta_A$ *: An assertion over* $\mathcal{V}_A$ *characterizing initial values added by A.*
- $\rho_A$ *: 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 (defined inside Defn. 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*.*

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.*

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 $\mathcal{V}^s_A$ are indeed defined by the base system, i.e., that $\mathcal{V}^{s_{ext}}_A \subseteq \mathcal{V}_B$, and that the domains of these variables, as defined in the aspects, are subdomains of the variables' domains as defined in the base.

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:*

- $\mathcal{V}_C = \mathcal{V}_B \cup \bigcup_{A \in \mathcal{A}} \mathcal{V}_A$
- $\theta_C$ *characterizes the set of initial states* $\theta_B \vee \bigvee_{A \in \mathcal{A}} \theta_A$
- $\rho_C$ *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* $\mathcal{V}^e_B \cup \bigcup_{A \in \mathcal{A}} \mathcal{V}^e_A$
- *Each run of* $\mathcal{C}$ *satisfies* $\bigwedge_{A \in \mathcal{A}} \mathcal{L}^s_A$*.*

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.*

## 3. A running example

Our running example is a printer management software, specified using a base and several aspects. The base describes a simple finite state machine over several system variables (`state`, `setup`, `print`) and a single environment

```
1  MODULE PrinterBase
2    VARENV -- environment variables (inputs)
3      newJob : boolean;
4    VAR   -- system variables
5      state : {ini,idle,work};
6      setup : {nil,warm,chk,done};
7      print : {nil,start,output,done};
8    ASSIGN
9      init(state) := ini;
10     init(setup) := nil;
11     init(print) := nil;
12
13     next(state) := case
14       state=ini & setup=done   : {ini,idle};
15       state=idle & newJob      : {idle,work};
16       state=work & print=done  : {work,idle};
17       1                        : state;
18     esac;
19     next(setup) := case
20       state=ini & setup=nil   : warm;
21       state=ini & setup=warm  : {warm,chk};
22       state=ini & setup=chk   : {chk,done};
23       state!=ini              : nil;
24       1                       : setup;
25     esac;
26     next(print) := case
27       state=work & print=nil     : start;
28       state=work & print=start   : output;
29       state=work & print=output  : done;
30       state=work & print=done    : nil;
31       1                          : print;
32     esac;
```

**Listing 1.** The code for the printer base system

controlled variable (`newJob`). Roughly, the printer starts in an `ini` state and following several steps can move to the `idle` state. Then, whenever a printing request is sent (the environment sets `newJob`), the system moves to `work` state, the request is printed (in several steps), and the system goes back to the `idle` state. The base serves as a blue print for the system to be. We show its code in List. 1.

Note that the base system is not deterministic. For example, when `state=ini` and `setup=warm`, it can either stay in `setup=warm` or move to `setup=chk` (representing 'check' phase) (line 21). As another example, when `state=idle` and the environment sets `newJob`, the system can move to `work` state or stay in `state=idle`. Such nondeterminism forms an abstraction for a set of states whose details are not yet specified. Still, the result of synthesis will be a concrete implementation, representing a deterministic controller.

Several features are defined on top of the base, and each is specified using a separate LTL aspect. We describe two of them here and leave two additional ones to Sect. 5.

The aspect `PrinterCancelJob` (List. 2) adds a cancel feature. Roughly, when the system is in `state=work` and the environment sets `cancel` to true (the user pressed the cancel button), the printer should immediately move to `state=idle`. More formally, this is specified in two parts. The `TRANS` section adds a transition to set the next

```
ASPECT PrinterCancelJob
  VARENV
    new cancel : boolean;
  VAR
    ext state : {idle, work};
    ext setup : {nil};
    ext print : {nil};
  TRANS
    cancel & next(state)=idle &
    next(print)=nil & next(setup)=nil;
  LTLSPEC
    [] ((cancel & state=work)
      -> next(state)=idle) &
    [] ((cancel & state=work)
      -> next(setup)=nil) &
    [] ((cancel & state=work)
      -> next(print)=nil);
```

**Listing 2.** The `PrinterCancelJob` LTL aspect

value of `state` to `idle` and of `setup` and `print` to `nil`. The `LTLSPEC` section is broken into three safety formulas: it must be globally true that immediately after `cancel & state=work` the system would satisfy `state=idle`, `setup=nil`, and `print=nil`. Thus, when `state=work` and `cancel` is true, this forces the system to follow the transition that was added in the `TRANS` section.

The aspect `PrinterGuarantees` (List. 3) forces the printer to (1) eventually complete its initialization and (2) eventually process jobs set by the environment. This is done by specifying liveness properties in the form of two response formulas: (1) globally, if `state=ini` then eventually `state=idle` and `setup=done`, and (2) globally, if `state=idle` and `newJob` is set, then eventually `state=work`. It could have been a better design to refactor these two guarantees into separate aspects (this would have no semantic consequences). We chose not to separate, so as to demonstrate that AspectLTL aspects can specify more than one liveness formula and that our traceability and debugging information is accurate: it points to specific formulas within the aspects.

Finally, the printer example is relatively small and simple. We use it to demonstrate two-way traceability and conflict

```
ASPECT PrinterGuarantees
  VARENV
    ext newJob : boolean;
  VAR
    ext state : {ini, idle, work};
    ext setup : {done};
  LTLSPEC
    // guarantee to finish ini.
    [] (state=ini
      -> <> (state=idle & setup=done)) &
    // guarantee to respond to newJob.
    [] ((state=idle & newJob)
      -> <> state=work);
```

**Listing 3.** The `PrinterGuarantees` LTL aspect

debugging, which are the focus of the paper. AspectLTL programs can be larger and much more complex.

## 4. Traceability

When an AspectLTL specification is realizable, traceability amounts to providing information that traces each allowed or forbidden transition in the resulting deterministic discrete system back to the aspects that have justified its presence or elimination. Below we formally define and demonstrate the concepts of positive and negative justifications and show how they are computed.

### 4.1 Positive justification

DEFINITION 7 (Positive justification).
*Let $\mathcal{S} = \langle \mathcal{B}, \mathcal{A} \rangle$ be an AspectLTL specification, let $\mathcal{C} = \langle \mathcal{V}_C, \theta_C, \rho_C \rangle$ be a* DS *that implements it, and let $t \in \rho_C$ be a transition in the implementation. We say that an aspect $A \in \mathcal{A}$ positively justifies $t$ iff $t \models \rho_A$. We say that the base $\mathcal{B}$ positively justifies $t$ iff $t \models \rho_B$. We denote the set of aspects (and base, if applicable) that positively justify $t$ by $pos(t)$.*

By definition of AspectLTL semantics, every transition in an implementation must be positively justified by the base or by at least one of the aspects, that is, $\forall t \in \rho_C, pos(t) \neq \emptyset$. A transition may be positively justified by several aspects.

For example, consider the specification $\mathcal{S} = \langle PrinterBase, \{PrinterCancelJob, PrinterGuarantees\}\rangle$. When in state `state=idle & cancel`, the synthesized implementation includes the self transition to remain in `state=idle`. Computing positive justifications shows us that this transition is justified by the base `PrinterBase` as well as by the `PrinterCancelJob` aspect. As another example, when in `state=work`, the synthesized implementation includes the transition to `state=idle` even when `print!=done`. This transition, however, is justified only by the `PrinterCancelJob` aspect.

### 4.2 Negative justification

Negative justification is defined for transitions that cannot be part of the implementation. We distinguish two kinds, explicit and implicit. Explicit negative justification happens when a safety formula disallows a transition. Implicit negative justification happens when a transition is disallowed because it leads to a losing state, i.e., a state from which the system cannot win the synthesis game. Formally:

DEFINITION 8 (Negative justification).
*Let $\mathcal{S} = \langle \mathcal{B}, \mathcal{A} \rangle$ be an AspectLTL specification, let $\mathcal{C} = \langle \mathcal{V}_C, \theta_C, \rho_C \rangle$ be a* DS *that implements it, and let $t$ be a transition from a state in the implementation to a losing state.*

- *We say that $t$ is explicitly negatively justified by the specification $\mathcal{S}$ iff (1) $t \models \rho_B \vee \bigvee_{A \in \mathcal{A}} \rho_A$ and (2) there exists $A \in \mathcal{A}$ such that $t \not\models \varphi_t$ where $\varphi_t$ is the transition part (safety formula) of $\mathcal{L}^s_A$.*

- *We say that $t$ is implicitly negatively justified by the specification $\mathcal{S}$ iff (1) $t \models \rho_B \vee \bigvee_{A\in\mathcal{A}} \rho_A$, (2) it is not explicitly negatively justified, and (3) it leads to a losing state, i.e., a state from which the system cannot guarantee to win the synthesis game (the environment can win).*

As an example, consider the specification $\mathcal{S} = \langle \mathit{PrinterBase}, \{\mathit{PrinterCancelJob}, \mathit{PrinterGuarantees}\}\rangle$. When `cancel` is set by the environment, the safety conjunct (lines 12-13) in `PrinterCancelJob` aspect `LTLSPEC` section, prevents `state=work` from staying in `state=work`, even if a `newJob` is set. Thus, this transition is explicitly negatively justified by `PrinterCancelJob`.

As another example, in the same specification, when a `newJob` is set in `state=idle`, the implementation cannot follow the transition that stays in `state=idle` (as allowed by `PrinterBase` at line 15) because this would lead to a losing state. For example if the environment chooses to reset the `newJob` and never set it again, the system will not be able to satisfy its response guarantee to eventually arrive to `state=work` (line 12-13 of `PrinterGuarantees`).

Note that not all transitions that are not in the implementation have to be negatively justified, explicitly or implicitly. Some such transitions may be not available in the implementation because they are not included in any of the transition relations defined by the base or the aspects, or because of choices made by the synthesis computation [19].

Finally, note that our definition of positive and negative justifications does not consider the possible effect of environment assumptions (see our remark at the end of Sect. 1). For example, in the presence of environment assumptions, additional traceability information that links the choice of transitions in the implementation to specific assumptions would be useful.

### 4.3 Computing two-way traceability

We compute positive justifications by iterating over the states of the implementation. For each transition from a state to a successor state, and for each of the `TRANS` sections of the aspects, we perform a single symbolic check to determine whether it is positively justified.

We compute explicit negative justifications by iterating over the states of the implementation. For each state in the implementation we identify the set of transitions that are allowed by the base or the `TRANS` sections of the aspects, but are blocked by safety formulas from some `LTLSPEC` sections. Computing implicit negative justifications is done by a similar symbolic operation, with respect to a 'safety formula' representing the set of transitions to all losing states (the symbolic representation of this set is a byproduct of the synthesis algorithm). To avoid exhaustive enumeration, we can either report the symbolic representation of each identified negative justification, or extract a witness.

Note that the traceability results point not only to the relevant aspects but also to their specific subformulas that participate in the justifications. We use the same results to provide traceability in the other way too: given a subformula from one of the aspects, we identify all transitions it has positively or negatively justified relative to a specific implementation. For example, our technique shows that the subformula in the `TRANS` section of the `PrinterCancelJob` aspect (lines 9-10), has positively justified many transitions in the implementation, e.g., a transition to `state=idle`, `print=nil`, and `setup=nil`, from many states where `state=work & cancel=true`, and from many states where `state=idle & cancel=true`.

```
ASPECT PrinterPause
  VARENV
    new pause : boolean; -- new input
  VAR
    ext print : {}; -- no value is assumed
  TRANS
    pause & print=next(print);
  LTLSPEC
    [] (pause -> print=next(print));
```

**Listing 4.** The `PrinterPause` LTL aspect

All traceability results are collected so that they can be reported and presented in the plug-in UI (see Sect. 7).

## 5. Extending the example

We now present additional aspects that we will use in Sect. 6.

The aspect `PrinterPause` (List. 4) adds a pause printing feature: When the user presses the pause button, the printer should stop changing its print status. The aspect includes a new environment variable `pause`, representing the input from a pause button. It is defined in two parts: the `TRANS` section adds a transition to preserve the value of `print` when `pause` is true, and the `LTLSPEC` section specifies a safety formula, to force the system to pause in this case.

The aspect `PrinterInkManagement` (List. 5) adds ink related functionality: updating the ink cartridge state and disallowing printing when no ink is left. It defines two new variables (1) an environment variable `fillCartridge` and (2) a system variable `ink` representing the level of ink in the cartridge. The aspect constrains the initial value of the new `ink` variable and the related transitions: either `fillCartridge` is set and the ink level is correspondingly set to 7 (representing the maximal ink level), or the level of `ink` is decreased by one after `print=output`. Moreover, the aspect forbids the printer from printing when `ink=0` (there is no ink left), using a safety formula.

## 6. Conflict debugging

Some specifications are unrealizable: no implementation exists that can satisfy all their required guarantees in all environments. Deciding realizability, however, is not enough. We are interested in generating a proof that shows how an adverse environment can force the system to violate its specification and identify the conflicting aspects to blame. The

```
ASPECT PrinterInkManagement
  VARENV
    new fillCartridge : boolean;
  VAR
    ext print : {start, output};
    new ink   : 0..7;
  LTLSPEC
    ink = 7 &
    [] (next(ink) = case
      fillCartridge         : 7;
      ink>0 & print=output  : ink - 1;
      1                     : ink;
    esac);
  LTLSPEC
    [] (! (ink=0 & print=start));
```

**Listing 5.** The `PrinterInkManagement` LTL aspect

proof we generate is interactive: it is an executable counter-implementation of the specification, which the engineer can not only manually statically analyze, but also 'play against'. The interactive, guided debugging session, illustrates to the engineer how, regardless of her choices, she will eventually end up violating one or more of the required specifications.

DEFINITION 9 (AspectLTL counter-implementation). *A discrete system $\mathcal{C} = \langle \mathcal{V}_C, \theta_C, \rho_C \rangle$ is a counter-implementation of an AspectLTL specification $\mathcal{S} = \langle \mathcal{B}, \mathcal{A} \rangle$ iff the following hold:*

- $\mathcal{V}_C = \mathcal{V}_B \cup \bigcup_{A \in \mathcal{A}} \mathcal{V}_A$
- $\theta_C$ *characterizes the set of initial states* $\theta_B \vee \bigvee_{A \in \mathcal{A}} \theta_A$
- $\rho_C$ *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* $\mathcal{V}^s_B \cup \bigcup_{A \in \mathcal{A}} \mathcal{V}^s_A$ *(i.e., the system variables rather than the environment variables as in Defn. 5)*
- *Each run of* $\mathcal{C}$ *satisfies* $\bigvee_{A \in \mathcal{A}} \neg \mathcal{L}^s_A$ *(i.e., falsifying one of the* LTL *specifications).*

LEMMA 1. *An AspectLTL specification is unrealizable iff it has a counter-implementation.*

The Lemma follows from the fact that the synthesis game is determined, that is, for every game instance there is a winning strategy for one of the players: either we can synthesize a controller or we can produce a counter-implementation.

The key to computing a counter-implementation is to set up a game with reversed roles. A solution to this game is an artifact representing an environment behavior that will eventually lead to violating the specification, no matter how the system will react. Such an artifact is a proof that the specification is unrealizable, as required.

Moreover, we accompany the counter-implementation with additional information. First, traceability information (as in the case of realizable specifications), which shows justifications to the transitions in the counter-implementation. Second, at each of the states in the counter-implementation,

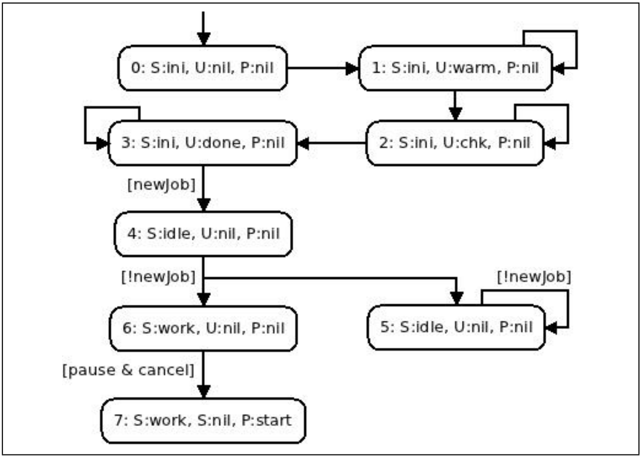


**Figure 1.** Debug information for Example I

we give pointers to the violated AspectLTL statements, i.e., to the specific liveness formulas, from the different aspects, that do not hold in the state. This information points to the exact aspects to blame.

We give two examples of unrealizable specifications with generated counter-implementations, as computed by the AspectLTL debugger.

## 6.1 Example I

As a first example, consider the AspectLTL specification $\mathcal{S} = \langle PrinterBase, \{PrinterCancelJob, PrinterGuar-antees, PrinterPause\} \rangle$. Is this specification realizable?

Our debugging technique identifies that this specification is unrealizable and generates a short counter-implementation, as illustrated in the diagram shown in Fig. 1. In the diagram we use abbreviated notation: `S` for `state`, `U` for `setup`, `P` for `print`. To make it more readable we use an abstraction: in the states we show only the values of the system variables. The values of environment controlled variables (inputs) for the next state are shown on its incoming transition as guards. A transition without a guard means that it is taken for a value that the environment chose from its possible winning strategies (i.e., the actual input value chosen by the environment makes no difference in this case for our example of proving unrealizability). We now explain the counter-implementation in detail.

From states 0 to 3, the system is in `state=ini`. In these states, the system guarantee to eventually reach `state=idle` (as specified in the `PrinterGuarantees` aspect, lines 9-10 of List. 3) is not met, so the environment allows the system to stay in these states as long as it wishes (with self transitions); if the system chooses to stay there forever, it will 'lose the game', because this guarantee will not be satisfied.

In state 4, the environment sets `newJob` to `true` (the environment sends a new print job). Now the system has two choices: either to start working on the new printing job (have `state=work` in state 6) or remain in `state=idle` in state 5. Both alternatives are, however, not good.

If the system chooses to stay in `state=idle`, the environment immediately resets `newJob` to false: the system is forced to stay forever in state 5, where the 'response to new job' guarantee specified in the `PrinterGuarantees` aspect (lines 12-13 of List. 3) is not met. If the system chooses to start working on the new job, the environment blocks it by setting the `cancel` and `pause` inputs to `true` (pressing the cancel and pause buttons at the same time!). Now, in state 7, the system reaches a deadlock, caused by a conflict between the safety formulas from the `PrinterCancelJob` and `PrinterPause` aspects: the first requires that the next state will have `print=nil` (lines 16-17 of the `PrinterCancelJob` aspect), while the second requires that in the next state the value of `print` will stay the same (line 9 of the `PrinterPause` aspect), i.e., in this case, remain `print=start`.

## 6.2 Example II

As a second, somewhat more complex example, consider the AspectLTL specification $\mathcal{S} = \langle PrinterBase, \{Printer\text{-}InkManagement, PrinterGuarantees, PrinterPause\}\rangle$. Is this specification realizable?

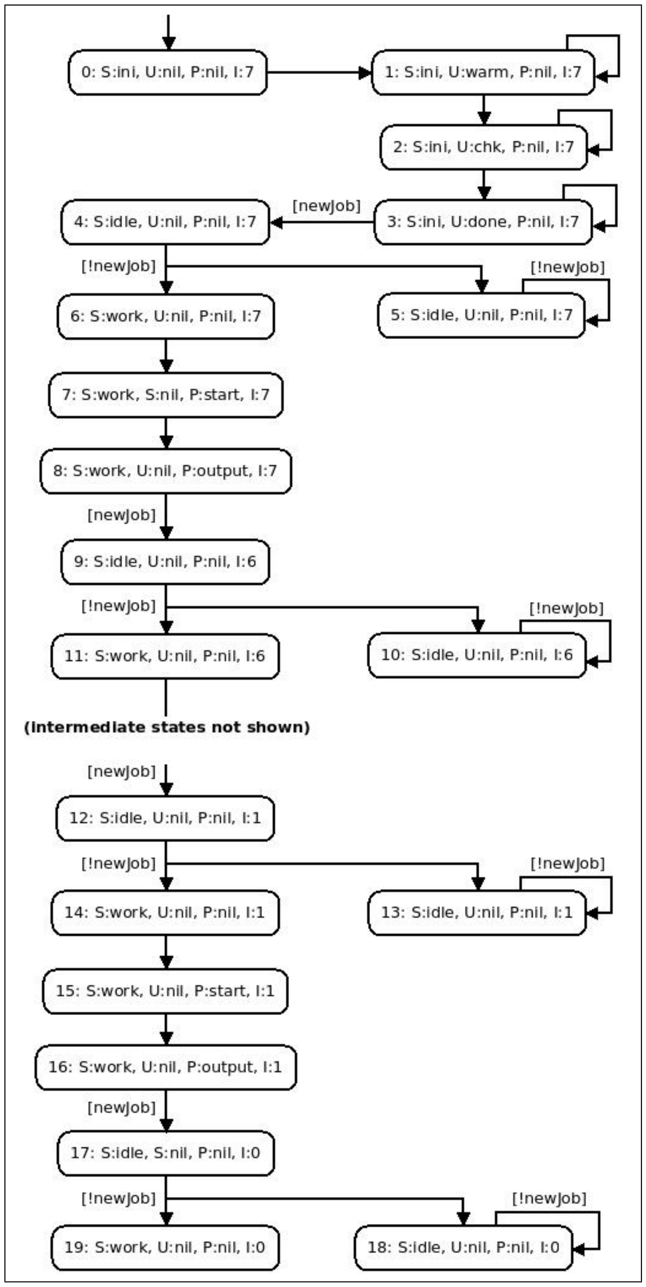
Our debugging technique identifies that this specification is unrealizable and generates a counter-implementation, as illustrated in the diagram shown in Fig. 2. Again we use abbreviated notation: `S` for `state`, `U` for `setup`, `P` for `print`, and `I` for `ink`. We now explain it in detail.

From states 0 to 3, as in the previous example, the system is in `state=ini`. In these states, the system guarantee to eventually reach `state=idle` (as specified in the `PrinterGuarantees` aspect, lines 9-10 of List. 3) is not met, so the environment allows the system to stay in these states as long as it wishes (with self transitions); if the system chooses to stay there forever, it will 'lose the game', because this guarantee will not be satisfied.

After reaching state 4, where the environment has sent a new printing job, the system needs to choose between two alternatives, either start working on the printing job (and move to state 6 where `state=work`) or stay in `state=idle` and move to state 5. Again, both alternatives are not good.

According to `PrinterGuarantees`, the new job set in state 4 requires that eventually `state=work`. If the system chooses not to start working immediately, the environment resets `newJob` and the system is forced to not reach `state=work`, forever, thus violating this guarantee. If the system chooses to start work immediately (and move to state 6), the run continues until the print job is done. As the environment tries to fail the system, it continues to set `newJob` whenever the system is back in `state=idle`, aiming to eventually reach a state where `ink=0`. Whenever `newJob` is set, the choice between working or staying in idle state is repeated (we do not show all repetitions in the diagram).

Finally, in state 17, there is no more ink, and the environment sets `newJob` for the last time. Again, if the system chooses not to handle this immediately and keep



0: S:ini, U:nil, P:nil, I:7
1: S:ini, U:warm, P:nil, I:7
2: S:ini, U:chk, P:nil, I:7
3: S:ini, U:done, P:nil, I:7
[newJob]
4: S:idle, U:nil, P:nil, I:7
[!newJob]
[!newJob]
5: S:idle, U:nil, P:nil, I:7
6: S:work, U:nil, P:nil, I:7
7: S:work, S:nil, P:start, I:7
8: S:work, U:nil, P:output, I:7
[newJob]
9: S:idle, U:nil, P:nil, I:6
[!newJob]
[!newJob]
10: S:idle, U:nil, P:nil, I:6
11: S:work, U:nil, P:nil, I:6
(Intermediate states not shown)
[newJob]
12: S:idle, U:nil, P:nil, I:1
[!newJob]
[!newJob]
13: S:idle, U:nil, P:nil, I:1
14: S:work, U:nil, P:nil, I:1
15: S:work, U:nil, P:start, I:1
16: S:work, U:nil, P:output, I:1
[newJob]
17: S:idle, S:nil, P:nil, I:0
[!newJob]
[!newJob]
18: S:idle, U:nil, P:nil, I:0
19: S:work, U:nil, P:nil, I:0

**Figure 2.** Debug information for Example II

`state=idle`, it is forced to stay in state 18 forever (the environment never sets `newJob` again) and thus lose the game by not satisfying the printing guarantee. If the system chooses to work on the job it moves to state 19 with `state=work`, the safety formula of line 15 in the `PrinterInkManagement` aspect prevents it from actually start printing (with `print=start`) and so it reaches a deadlock.

### 6.2.1 Unrealizable core

Interestingly, note that in this counter-implementation the environment did not 'use' the input `pause`, although the `PrinterPause` aspect is included in the specification. Indeed, the traceability information that we compute shows that the specifications of the `PrinterPause` aspect (the added transition and the safety formula) neither positively justified any of the transitions in the counter-implementation nor removed any negatively justified transitions that could have been defined as outgoing from the states in the counter-implementation. Thus, the traceability information shows that `PrinterPause` was not necessary in the proof for unrealizability: the combination of `PrinterInkManagement` and `PrinterGuarantees` is, by itself, unrealizable.

Our approach can identify such unnecessary aspects in the unrealizability proof, after a counter-implementation is computed. However, our current technique is not guaranteed to find a minimal unrealizable subset of the specification. This relates to the problem of finding an unrealizable core. See our discussion of future work in Sect. 9.

### 6.3 Computing a counter-implementation

Computing the counter-implementation is done by solving a Rabin game where the environment tries to falsify at least one of the system's guarantees, following the fixpoint algorithms described in [13, 18]. Roughly, the algorithm starts from the set of states from which the system has no valid possible successors. It then iterates 'backwards' by adding states from which the environment can either force the system to (1) reach previously found losing states, or (2) constantly violate one of the system's guarantees (each set of states where the guarantee is constantly violated, is computed using another nested fixpoint).

The fixpoint is reached when no additional losing states can be found. If there exists an environment initial choice for which all the system's initial choices are in the computed set, then the specification is unrealizable. A counter-implementation can be constructed from the intermediate values of the fixpoint computation (see [13, 19]), while pointing to the system's guarantee that the strategy is trying to falsify at each state of the counter-implementation. In the resulting counter-implementation, as presented to the user, each state is annotated with (1) the system's guarantee that the strategy is "currently" trying to falsify, and (2) the related traceability information as described in Sect. 4.

### 6.4 Environment assumptions

As mentioned at the end of Sect. 1, most of our discussion in this paper does not consider environment assumptions. In the context of conflict debugging, however, the presence of environment assumptions is significant: as assumptions limit the environment to certain behaviors, their addition may make a previously unrealizable specification realizable.

Consider the example presented in Sect. 6.1. Adding a simple assumption that 'prevents' the user from pressing the cancel and pause buttons at the same time, formally adding `[](!(pause & cancel))` as an assumption, renders the specification realizable. Indeed, in this case, the AspectLTL plug-in successfully synthesizes a controller and generates an implementation.

Consider the example presented in Sect. 6.2. Adding a simple assumption, specifying that if the ink is empty, then a refill must occur in the following step, formally adding `[](ink=0 -> next(fillCartridge))` as an assumption, renders the specification realizable. As in the previous example, in this case, the AspectLTL plug-in successfully synthesizes a controller and generates an implementation.

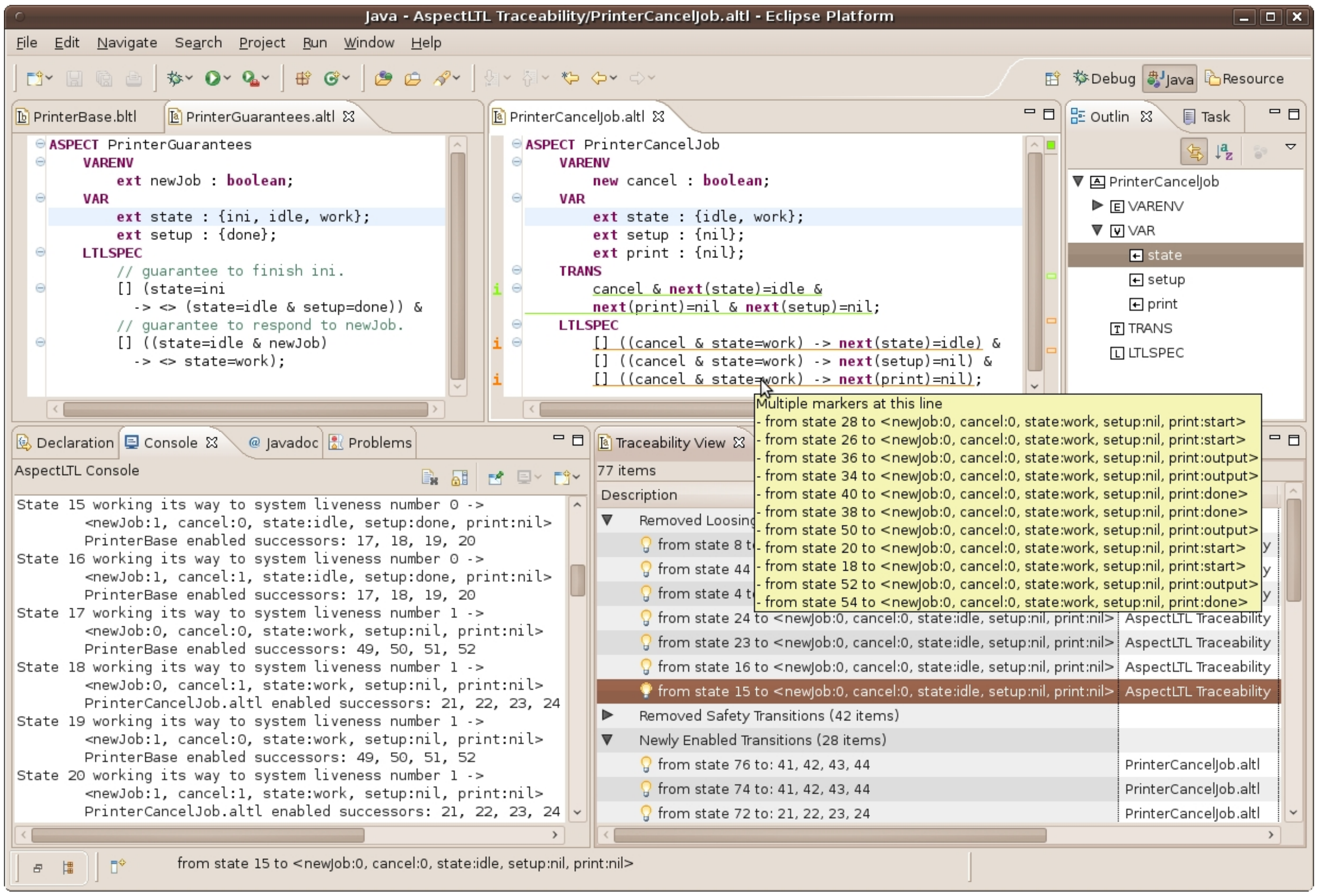
Interestingly, however, note that the assumption we suggested to add to the first example, `[](!(pause & cancel))`, involves two environment variables that belong to different LTL aspects, `PrinterCancelJob` and `PrinterPause`! This assumption is thus not local to any of the aspects alone and its use may be viewed as violating the very idea of separation of concerns. We leave this for discussion in future work.

## 7. Implementation and Evaluation

We have implemented our ideas and integrated them into the AspectLTL plug-in. The implementation is based on the APIs of JTLV [22], a framework for the development of formal verification algorithms using BDD-based mechanism. The plug-in includes front-end editors with syntax highlighting for AspectLTL, on-the-fly parsing and quick fixes, outline, auto-completion, and views and markers to mark traceability information. The plug-in together with related documentation are available from our website [2]. All examples shown in this paper and mentioned in the evaluation below are available with the plug-in, so that all experiments can be reproduced. We encourage the interested reader to try them.

Fig. 3 shows a screen capture from the AspectLTL plug-in, displaying a tracing session of the specification $\mathcal{S} = \langle PrinterBase, \{PrinterCancelJob, PrinterGuarantees\}\rangle$. Two aspects are shown in the main editor windows, with green markers highlighting subformulas that induce positive justifications and orange markers highlighting subformulas that induce explicit negative justifications. Note that the second safety formula of the aspect `PrinterCancelJob` is not marked, meaning that it does not induce any justifications. The tooltip over the orange marker relates to the third safety formula and presents the transitions that it negatively justifies, that is, it shows traceability from aspect specification to program behaviors. The complete list of justifications is shown in the lower right pane; clicking each item in the table leads to its related subformula in one of the aspects, that is, traceability from program behavior to the specification. The textual representation of the implemented controller is shown on the lower left (also annotated with per-state traceability information).

The plug-in supports a graph-based view of the implementation (or counter-implementation), similar to the graphs shown in Fig. 1 and Fig. 2. The graph unfolds dynamically, according to the engineer's choice of actions at each step.



```
ASPECT PrinterGuarantees
    VARENV
        ext newJob : boolean;
    VAR
        ext state : {ini, idle, work};
        ext setup : {done};
    LTLSPEC
        // guarantee to finish ini.
        [] (state=ini
          -> <> (state=idle & setup=done)) &
        // guarantee to respond to newJob.
        [] ((state=idle & newJob)
          -> <> state=work);
```

```
ASPECT PrinterCancelJob
    VARENV
        new cancel : boolean;
    VAR
        ext state : {idle, work};
        ext setup : {nil};
        ext print : {nil};
    TRANS
        cancel & next(state)=idle &
        next(print)=nil & next(setup)=nil;
    LTLSPEC
        [] ((cancel & state=work) -> next(state)=idle) &
        [] ((cancel & state=work) -> next(setup)=nil) &
        [] ((cancel & state=work) -> next(print)=nil);
```

```
Multiple markers at this line
- from state 28 to <newJob:0, cancel:0, state:work, setup:nil, print:start>
- from state 26 to <newJob:0, cancel:0, state:work, setup:nil, print:start>
- from state 36 to <newJob:0, cancel:0, state:work, setup:nil, print:output>
- from state 34 to <newJob:0, cancel:0, state:work, setup:nil, print:output>
- from state 40 to <newJob:0, cancel:0, state:work, setup:nil, print:done>
- from state 38 to <newJob:0, cancel:0, state:work, setup:nil, print:done>
- from state 50 to <newJob:0, cancel:0, state:work, setup:nil, print:output>
- from state 20 to <newJob:0, cancel:0, state:work, setup:nil, print:start>
- from state 18 to <newJob:0, cancel:0, state:work, setup:nil, print:start>
- from state 52 to <newJob:0, cancel:0, state:work, setup:nil, print:output>
- from state 54 to <newJob:0, cancel:0, state:work, setup:nil, print:done>
```

```
State 15 working its way to system liveness number 0 ->
        <newJob:1, cancel:0, state:idle, setup:done, print:nil>
        PrinterBase enabled successors: 17, 18, 19, 20
State 16 working its way to system liveness number 0 ->
        <newJob:1, cancel:1, state:idle, setup:done, print:nil>
        PrinterBase enabled successors: 17, 18, 19, 20
State 17 working its way to system liveness number 1 ->
        <newJob:0, cancel:0, state:work, setup:nil, print:nil>
        PrinterBase enabled successors: 49, 50, 51, 52
State 18 working its way to system liveness number 1 ->
        <newJob:0, cancel:1, state:work, setup:nil, print:nil>
        PrinterCancelJob.altl enabled successors: 21, 22, 23, 24
State 19 working its way to system liveness number 1 ->
        <newJob:1, cancel:0, state:work, setup:nil, print:nil>
        PrinterBase enabled successors: 49, 50, 51, 52
State 20 working its way to system liveness number 1 ->
        <newJob:1, cancel:1, state:work, setup:nil, print:nil>
        PrinterCancelJob.altl enabled successors: 21, 22, 23, 24
```

```
▼ Removed Loosing
    from state 8 t                                                        y
    from state 44                                                         y
    from state 4 t                                                        y
    from state 24 to <newJob:0, cancel:0, state:idle, setup:nil, print:nil>  AspectLTL Traceability
    from state 23 to <newJob:0, cancel:0, state:idle, setup:nil, print:nil>  AspectLTL Traceability
    from state 16 to <newJob:0, cancel:0, state:idle, setup:nil, print:nil>  AspectLTL Traceability
    from state 15 to <newJob:0, cancel:0, state:idle, setup:nil, print:nil>  AspectLTL Traceability
▶ Removed Safety Transitions (42 items)
▼ Newly Enabled Transitions (28 items)
    from state 76 to: 41, 42, 43, 44                                         PrinterCancelJob.altl
    from state 74 to: 41, 42, 43, 44                                         PrinterCancelJob.altl
    from state 72 to: 21, 22, 23, 24                                         PrinterCancelJob.altl
```

```
from state 15 to <newJob:0, cancel:0, state:idle, setup:nil, print:nil>
```

**Figure 3.** A screen capture of AspectLTL plug-in, displaying traceability information for the specification $\mathcal{S} = \langle PrinterBase, \{PrinterCancelJob, PrinterGuarantees\}\rangle$.

An example screenshot showing this graph is available from our website.

To emphasize the dynamic nature of AspectLTL debugging, a movie showing a typical debugging session that shows an execution of a generated, interactive counter-implementation program, is available from our website.

AspectLTL synthesis is based on GR(1) synthesis [19], whose complexity is cubic in number of states of the implementation, measured in symbolic steps. The symbolic algorithm allows it to scale well, at least up to medium size specifications [4]. While checking realizability / unrealizability is completely symbolic, the computation of a concrete implementation (or counter-implementation) requires the enumeration of states, which in some cases is much slower and does not scale well (specifically when dealing with data rather than control). Computing the additional traceability information is linear in the number of states and transitions in the implementation. The complexity of computing a counter-implementation is the same as that of synthesis.

To give a sense of the feasibility of AspectLTL synthesis, with and without traceability information, we report initial quantitative results from the performance of AspectLTL realizability checking and implementation generation.[3] The experiments were performed on a regular computer, Intel Dual Core CPU, 2.4 GHz, with 4 GB RAM, running 64-bit Linux. Running times are reported in milliseconds.

Table 1 shows results of experiments on the `Printer` system and on two additional systems: `ExamService`, described in detail in [17], and `Traffic`, a two-cars race problem. These examples, and some additional ones, are available with the plug-in. For each system we report on several configurations. For each configuration we report on the number of aspects, the number of LTL specs (the total number of safety and liveness conjuncts) + the number of transition disjuncts, the state space of the specification, whether the specification was realizable or not, the time for checking realizability, the size of the implementation (or counter-implementation), the time for computing the implementation (or counter-implementation), and the additional time of computing traceability information. Additional performance

[3] We show the concrete performance results in order to give a general, rough idea about the feasibility of using AspectLTL. We have not made special efforts to optimize our implementation and do not consider the exact values shown to be important: optimizing the performance of AspectLTL synthesis is outside the scope of this paper.

| Specification | # Aspects / # Specs (+ T) | State space | Realizable? | Deciding (ms) | Impl. size | Impl. gen. time (ms) | Traceab. time (ms) |
|---|---|---|---|---|---|---|---|
| PrinterBase weaved with: CancelJob (CJ), Guarantees (G), Pause (P), InkManagement (IM). | | | | | | | |
| CJ + G | 2/5 (+1) | $2^{12}$ | true | 34 | 77 | 81 | 57 |
| CJ + G + P | 3/6 (+2) | $2^{13}$ | false | 34 | 9 | 48 | 2 |
| CJ + G + P + IM | 4/8 (+2) | $2^{17}$ | false | 62 | 9 | 88 | 5 |
| CJ + G + IM | 3/7 (+1) | $2^{16}$ | false | 53 | 43 | 75 | 27 |
| ExamService weaved with: Tuition (T), AvailabilityBug (AB), Availability (A), FailuresLogging (FL), AllowQuitting (AQ), ExamProtection (EP), ExamCounter (EC). | | | | | | | |
| T + AB | 2/2 (+1) | $2^{10}$ | false | 10 | 10 | 13 | 1 |
| T + A + FL | 3/3 (+1) | $2^{12}$ | true | 36 | 64 | 58 | 11 |
| T + A + FL + AQ + EP | 5/6 (+2) | $2^{17}$ | true | 34 | 272 | 240 | 81 |
| T + A + AQ + EP + EC | 5/8 (+3) | $2^{18}$ | true | 57 | 1760 | 7128 | 854 |
| TrafficBase weaved with: Goal (G), Safety (S), Obstacle3 (O3), Obstacle4 (O4), ExtraSafety (ES), ReverseGear (RG). | | | | | | | |
| G + S | 2/2 (+0) | $2^{16}$ | true | 65 | 31 | 68 | 8 |
| G + S + ES + O3 + O4 | 5/6 (+0) | $2^{16}$ | false | 149 | 1 | 243 | 1 |
| G + S + ES + O3 + O4 + RG | 6/7 (+1) | $2^{16}$ | true | 455 | 37 | 459 | 9 |

**Table 1.** Results from running AspectLTL on several examples, checking realizability, generating an implementation (counter-implementation), and computing traceability information (see Sect. 7)

results are available in supporting materials [2]. The reported times do not include the marking in the eclipse UI.

The results suggest the following observations. First, checking for realizability is sometimes much faster and scales better than generating an implementation. Thus, we recommend the frequent use of realizability checking, without implementation generation (in the plug-in these features are intentionally separated). It is best if the engineer computes the implementation only when she wants to interactively execute the generated code, or when she finds that the specification is unrealizable. Second, the additional computation of traceability information is sometimes costly (proportional to the implementation size) but still does not carry a dramatic overhead on top of the implementation generation time. If it becomes too slow, one may allow to compute it on-demand for user-selected state/transition. Finally, overall, AspectLTL synthesis, with and without traceability information, seems to perform in acceptable time for all our examples. Larger examples may be slower and require the development of additional techniques, e.g., an incremental approach. We leave this for future work.

## 8. Related Work

We now discuss several related studies in the areas of traceability, feature interaction, and model-checking of aspects.

Borger et al. [3] present runtime, dynamic aspect traceability for AspectJ, by inspecting the stack to discover which pointcut causes a certain advice. Support for static traceability is provided, e.g., by AJDT [1], where in the source code view, the IDE shows which aspects (may) affect certain base artifacts and statements. In the context of AspectLTL, traceability is very different, in particular because the language is declarative and not imperative: the high-level motivation of relating aspect code artifacts with the concrete behavior they induce is similar, but the setup and technologies used are completely different.

Checking the realizability of AspectLTL specifications and the debugging of unrealizable ones, are related to works that identify feature interactions or use model-checking to discover aspect interference. We mention some of the most relevant ones below.

Felty et al. [6] present the specification of features using LTL and a model-checking based method to automatically detect conflicts between features. Realizability is approximated by model-checking only a given scope: if a conflict is detected, it is a real conflict, and a counter example is provided. If no conflict is detected, the result is inconclusive, i.e., conflict detection is sound but incomplete. In contrast, our realizability checks are sound and complete. Moreover, when a specification is realizable, we provide a correct-by-construction implementation, which is not available in [6].

Many approaches specify features using state machines and consider feature interaction as part of their composition (see, e.g., [8, 9, 15]). To the best of our knowledge, none considers a symbolic, declarative representation like the one used in AspectLTL and most compositions are not sound and complete like AspectLTL synthesis. We could not find a description of traceability and debugging features similar to the ones presented in our work.

Katz and Katz [10] present incremental aspect interference analysis. The work models AspectJ-like aspects using an SMV-like format. Detection of various interferences is done using model checking. In case of conflict, a counter example is provided. Our work uses an SMV-like format to specify aspects. However, rather than using the specification for model checking, we use it as an input for synthesis. In case of unrealizability, we provide a counter-implementation.

Li et al. [15] present a methodology that views cross-cutting features as independent modules and verifies them against CTL properties as open systems. Features consists of state machines and composition is done by connecting them via transitions specified through interfaces. The work supports the detection of undesirable feature interactions. Support for traceability and debugging is not described.

In contrast, AspectLTL aspects are defined in a symbolic and declarative manner. Our method is fundamentally different: it not only solves the possible conflicts or interferences between the specified aspects (if indeed a solution to these conflicts exists) but also produces an executable correct-by-construction implementation. If a solution does not exist, we generate a counter-implementation, annotated with the traceability information that uncovers reasons for unrealizability.

## 9. Conclusion and Future Work

We presented two-way traceability and conflict debugging techniques for AspectLTL and demonstrated them on a running example. To support two-way traceability, we use symbolic operations that check for intersections between the transitions that can or cannot be taken and the formulas defined in the LTL aspects. To support debugging of unrealizable specifications we reverse the roles of the system and the environment in the synthesis game, and use the winning strategy of the environment to produce a counter-implementation, that is, an interactive program, whose runs show exactly how any generated system can be forced by an (adverse) environment to violate the specifications. We combine traceability and debugging to point at the aspects to blame. The techniques provide important support for the development of systems using AspectLTL, making its use more accessible and informative. The ideas are implemented in the AspectLTL plug-in, available from [2].

One future work direction deals with the computation of an unrealizable core. Given an unrealizable AspectLTL specification, an unrealizable core is a minimal unrealizable subset of the specification. An unrealizable core is useful in debugging, as it better identifies and isolates the causes of failures and enables the generation of smaller counter-implementations. Some recent works have considered the computation of unrealizable cores in the context of LTL (GR(1)) synthesis (see, e.g., [5]). However, computing an unrealizable core for AspectLTL specifications is particularly challenging due to the non-monotonic nature of the language: each aspect may not only restrict the possible behaviors (in its `LTLSPEC` sections) but also add new behaviors (in its `TRANS` sections). Another future work is to investigate how our approach to traceability and debugging can be applied to other aspect languages, e.g., AspectJ, using abstractions similar to the ones of [7, 14], or more generally, to other feature composition frameworks (e.g., [8, 9, 15]), where, we believe, similar two-way traceability and conflict debugging support could be very useful.

## References

[1] AspectJ Development Tools. http://www.eclipse.org/ajdt/.

[2] AspectLTL website. http://aspectltl.ysaar.net/.

[3] W. D. Borger, B. Lagaisse, and W. Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *AOSD*, pages 173–184, 2009.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[5] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *VMCAI*, 2008.

[6] A. P. Felty and K. S. Namjoshi. Feature specification and automated conflict detection. *ACM Trans. Softw. Eng. Methodol.*, 12(1):3–27, 2003.

[7] M. Goldman, E. Katz, and S. Katz. MAVEN: modular aspect verification and interference analysis. *Formal Methods in System Design*, 37(1):61–92, 2010.

[8] J. D. Hay and J. M. Atlee. Composing features and resolving interactions. In *SIGSOFT FSE*, pages 110–119, 2000.

[9] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Software Eng.*, 24(10):831–847, 1998.

[10] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *FOAL*, pages 29–38, 2008.

[11] S. Katz. Aspect categories and classes of temporal properties. In *T. Aspect-Oriented Softw. Dev. I*, pages 106–134, 2006.

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

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

[14] S. Krishnamurthi and K. Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2), 2007.

[15] H. C. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *SIGSOFT FSE*, pages 89–98, 2002.

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

[17] S. Maoz and Y. Sa'ar. AspectLTL: An aspect langauge for LTL specifications. In *AOSD*, pages 19–30, 2011.

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

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

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

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

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

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