

# SpeAR Design Document

## Contents

1.	Purpose	3
2.	Intended Reader and Prerequisites	3
3.	Tool Architecture Overview	4
4.	Grammar and Parsing	6
4.1.	File Importing and Conflicts	6
5.	Validation	7
5.1.	Unique Names	7
5.2.	References are Acyclic	7
5.3.	Type Checking	7
5.4.	Unit Checking	8
5.5.	Unused Variable Identification	8
5.6.	Unsupported Construct Highlighting	9
6.	Document Representation	10
7.	Transformations	11
7.1.	Replacing Abstract Types	11
7.2.	Propagate Predicates	11
7.3.	Creation of User Namespace	13
7.4.	Remove Composite References	13
7.5.	Replace Shorthand Records	14
7.6.	Normalize Operators	14
7.7.	Removing Syntactic Sugar	15
7.8.	Expanding “In” Expressions	16
7.9.	Generating UFC Obligations	16
7.10.	Replace Specification Calls	17
7.11.	Uniquify Normalized Calls	17
8.	Translation to Lustre	19
8.1.	Internal Representation	19
8.2.	Type and Expression Mapping	20
8.3.	Translation of Normalized Specification Calls	21

8.4.	Observer Properties	22
8.5.	Translation Commonality	23
8.6.	Patterns Analysis Translation	25
9.	Results Presentation	27
9.1.	Logical Entailment Analysis Results	27
9.2.	Traceability	28
9.3.	Logical Consistency	29
9.4.	Realizability	30
10.	References	32

## 1. Purpose

This document is intended to capture the detailed design of the SpeAR requirements analysis tool, described in *SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements* [1].

## 2. Intended Reader and Prerequisites

The intended reader for this document is an experienced user of SpeAR who wishes to develop a deeper understanding of how the tool is built.

Those who wish to modify or enhance SpeAR should be familiar with the following items or publications.

- Java programming language (Java 8 constructs are often used in the code base) [2]
- Eclipse Java IDE [3]
- Xtext language development framework [4]
- Lustre synchronous dataflow language [5]
- JKind model checking tool [6]
- The jRealizability tool [7]
- Inductive Validity Cores [8]

### 3. Tool Architecture Overview

SpeAR is an Eclipse-based formal requirements development and analysis environment. Implemented in Java, SpeAR utilizes the Xtext framework to define and generate language artifacts. Xtext takes a language grammar and configuration file as input and generates the language artifacts necessary to parse and translate SpeAR files to Lustre. Xtext also provides the capability to execute custom validation checks for code opened in the editing environment. The result is a real-time integrated development environment for building and analyzing formal requirements specifications.

The SpeAR tool architecture has two components. The first component is the real-time development environment that provides feedback to users as they develop specifications. This environment, shown in the shaded blue region of Figure 1, will run a series of analyses every time the specification changes to ensure that it is correct with respect to type usage, units usage, and other well-formedness checks. It also checks to see if the specification can be analyzed to ensure it contains no errors that would prevent the underlying model checker, JKind, to operate. The second component is the translation to the Lustre modeling language, which is analyzable by JKind. This element is shaded in green in Figure 1. Translation occurs in a multi-step process. First, a copy of all the relevant workspace files, encapsulated in a Java class named *Document*, is created. This is discussed further in Section 6. Then, the document is passed through a series of transformations to prepare it for translation. This is discussed in Section 7. Next, the transformed document is translated into an intermediate representation, and then to Lustre, which is discussed in Section 8. Finally, the Lustre file is analyzed by JKind, and the results are collected and mapped to SpeAR to be presented in the editing environment. This element is shaded in violet in Figure 1.

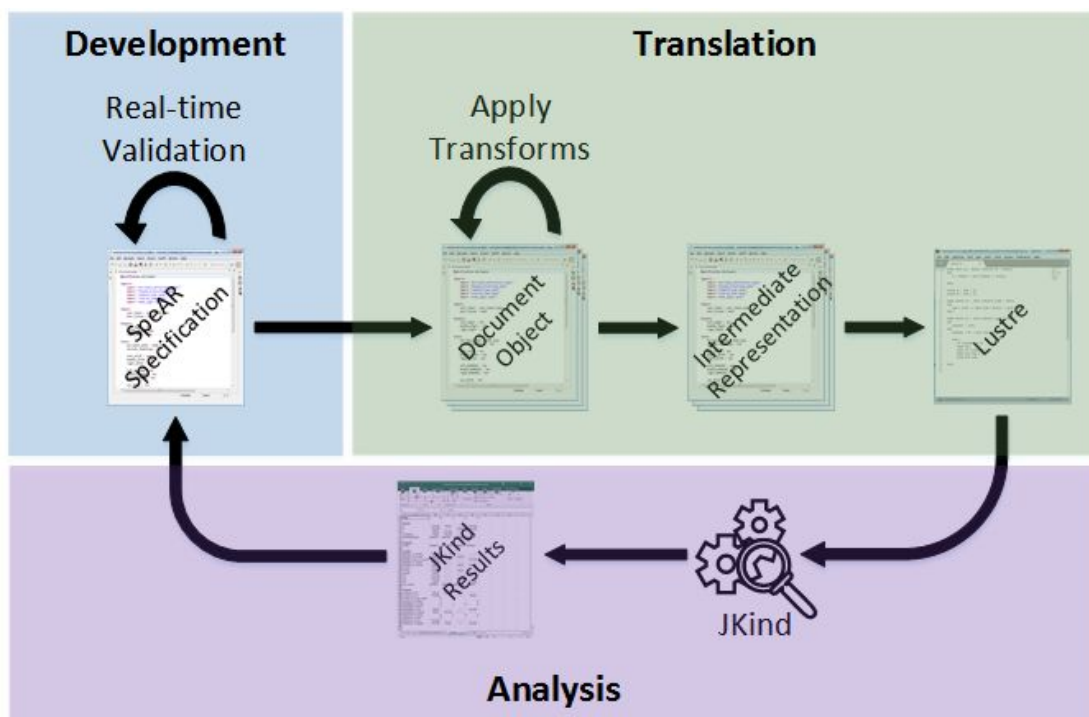


Figure 1 – SpeAR tool architecture

The remainder of this document discusses how each action shown in Figure 1 is accomplished with links to the relevant code in the repository.

## 4. Grammar and Parsing

The SpeAR language is defined in the Xtext grammar language [9]. Once the language is defined, Xtext generates a language parser that is used to create a semantic model of the user's SpeAR file. This model captures, in memory, all of the behaviors described in the user's SpeAR specification. This model can be used as a springboard to translate into Lustre, enabling SpeAR's different analyses.

The grammar for SpeAR's input language is available at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear/src/com/rockwellcollins/Spear.xtext>

### 4.1. File Importing and Conflicts

Unlike Lustre, SpeAR allows file importing. This can create naming conflicts when a file imports another file with an identical name. Conflicts are resolved by using the local file's name over the imported file's name. Figure 2 walks through an example that demonstrates how a simple naming conflict between two specifications is handled in SpeAR.

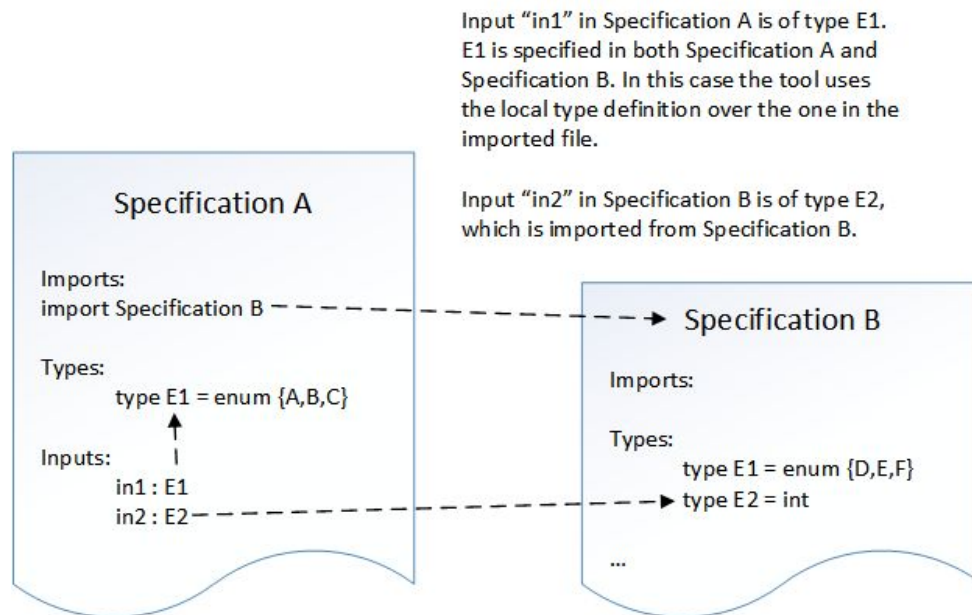


Figure 2 - Conflicting namespace resolution

In addition, SpeAR provides analyses to ensure that users have not introduced a circular import chain. In Figure 2 SpeAR would produce an error if Specification B imported Specification A, as Specification A already imports Specification B. Imports also only work in a single direction. Specification A has access to the types, constants, patterns, and the specification B itself. Specification B does not have access to the entities Specification A.

## 5. Validation

Parsing ensures the input to SpeAR satisfies its grammar; however, additional checks are needed to ensure it is well formed and compatible with the analysis target language, Lustre. For example, Lustre does not allow cyclic references between types, constants, and node calls. To ensure this does not occur in SpeAR, it is necessary to check that every element that could be translated into one of those Lustre constructs (types, macros, constants, specifications) does not have cyclic references.

The following sections describe the validations performed by SpeAR, the justification for each, and insight into the kinds of issues that could occur if the validation was not performed.

### 5.1. Unique Names

This validation ensures that each element of the grammar has a unique identifier. Naming conflicts in Lustre will cause the underlying analysis tool, JKind, to fail. This is prevented by performing analyses to identify naming conflicts in real-time so they may be corrected by the user prior to analysis.

The code for SpeAR's unique naming analysis is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear/src/com/rockwellcollins/validation/NamesUnique.java>

### 5.2. References are Acyclic

This validation ensures that types, constants, macros, patterns, specifications, and other referential SpeAR elements are not self-referential. This includes checking through a chain of references to ensure that two or more elements are not mutually self-referential. JKind fails when certain elements are self-referential, and thus this validation is necessary to ensure the final file can be successfully analyzed.

The code for identifying cyclic references is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear/src/com/rockwellcollins/validation/AcyclicValidator.java>

### 5.3. Type Checking

SpeAR's type checker is both structural and nominal. Array types are equivalent based on structure; that is, if both array types have the same base type and are of the same length, then they are equivalent. Record types are equivalent only if they reference the same record type name, even if structurally equivalent. Abstract types are equated nominally, as there is no structural information available. Finally, predicate subtypes are equated based on the base type. If a predicate subtype's base type references another predicate subtype, the type-checker recursively navigates down to the first non-predicate subtype and uses that as a basis for comparison.

Due to limitations in Lustre, only primitive types, array types, and record types are translated into Lustre. Other types must be processed to make them suitable for Lustre. Removal of abstract types is performed by the Replace Abstract Types transformation, described in Section 7.1. Variables typed with predicate subtypes have additional assertions or proof obligations on them, introduced by the Propagate Predicates transformation. Finally, variables typed by predicate subtypes are typed in Lustre as the base type of the predicate subtype.

It is necessary to perform type checking to ensure the generated Lustre file is analyzable by JKind. It also provides feedback to the user on whether or not their SpeAR specifications are coherent and well formed.

The code for SpeAR's type checking is at

<https://github.com/lgwagner/SpeAR/tree/master/com.rockwellcollins.spear/src/com/rockwellcollins/spear/typing>

## 5.4. Unit Checking

Similar to SpeAR's type checker, the unit checker confirms that various elements within a model are in agreement with respect to units. For example, if the user adds a variable with units of meters to a variable with units of feet, SpeAR flags this as an error. SpeAR also performs dimensional analysis over arithmetic operations, and is able to ensure the result of a mathematical operation is in unit agreement with the variable it is stored in. This allows for convenient manipulation of different types (such as position, velocity, and acceleration) through arithmetic, while the tool ensures correct dimensionality is preserved. For an example of the use of units, see

<https://github.com/lgwagner/SpeAR/wiki/Units>

Arithmetic over units is computed by creating a map keyed by the unit identifiers with values representing the exponent of a unit in a given expression. As arithmetic is performed, units can be multiplied or divided, thus modifying the exponent of the unit in the map, to provide the simplified version. If the exponent of a unit is 0 after a computation, it is removed from the map because it has been cancelled out. As an example, consider arithmetic over a variable with units of meters. Table 1 shows the representation of the map object used to compute the units after multiplying the entities of units meters (m), which is then divided by an entity of units (seconds (s)/m<sup>2</sup>), then multiplied by an entity of units (m<sup>3</sup>s), and then finally divided by units (ms<sup>2</sup>), yielding a final result of units of meters (m).

Initial Units (m)	Divide by units (m <sup>2</sup> /s)	Multiply by units (m <sup>3</sup> s)	Divide by units (ms <sup>2</sup> )
m → 1	m → -1 s → 1	m → 2 s → 2	m → 1 s → 0

Table 1 - Units computations through a map object

Units checking is performed as a static analysis over the SpeAR specification. It is not required or necessary to run formal analysis and the units are not propagated into the Lustre translation. The code for SpeAR's units checking can be found at

<https://github.com/lgwagner/SpeAR/tree/master/com.rockwellcollins.spear/src/com/rockwellcollins/spear/units>

## 5.5. Unused Variable Identification

SpeAR provides an analysis to provide warnings when specification variables are not used. This is provided so users can understand which variables in a SpeAR specification are referenced. It can be helpful for identifying unused specification elements so that they may be removed or additional requirements can be added to address them. It can also be distracting during development of a new specification, thus an option to suppress them can be found in the SpeAR options menu.

Unused variables do not prevent SpeAR from performing logical entailment, consistency, or realizability analysis. If variables are unused, they are modeled as unconstrained variables and are allowed to take on any legal value for their assigned type.

The code for SpeAR's unused variable analysis is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear/src/com/rockwellcollins/validation/VariablesAreUsedValidator.java>

## 5.6. Unsupported Construct Highlighting

SpeAR enforces numerous limitations on well-formed grammar beyond the static analyses mentioned in the previous section. Many of these checks are to ensure the user is not using a SpeAR language element in a way that would prevent successful analysis. Some examples include:

- The use of specification calls inside of patterns
- The use of specification calls inside of macros
- Using specification calls in a way that is not supported in Lustre

The code for these analyses can be found at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear/src/com/rockwellcollins/validation/IllegalAnalysisValidations.java>

Further, some validations ensure that the specification is well formed per agreed-upon rules by the developer and stakeholders. These include:

- Specification names match base filenames
- Constants are either constant literals or derived wholly from constant literals
- "Initially" expressions are not embedded within other expressions
- Section headers are not duplicated
- Observer and UFC (see Section 7.9 for more information) flags on requirements are used on the correct elements
- Nonlinear operations are only used to the extent to which they are supported by the underlying solver
- Definitions only import other Definitions files, not Specifications

The code for these analyses is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear/src/com/rockwellcollins/validation/SpearJavaValidator.java>

## 6. Document Representation

Once a specification passes validation and is free from errors, it is then analyzable by SpeAR's three types of analyses: logical entailment, consistency, and realizability. The translation approach utilizes transformations to modify the SpeAR specification to optimize it for translation to Lustre. Since each transformation will change the syntax of the SpeAR specification and its imported files, it is necessary to copy those files so the user's original files remain intact. SpeAR also supports file imports, so it is also necessary to modify imported files. These complicating factors make it necessary to create an intermediate representation that captures all of the files referenced by a SpeAR specification, and to



copy it, prior to applying transformations. This intermediate representation is captured in the Document class in the SpeAR code.

The Document class tracks several pieces of data required to translate a SpeAR specification to Lustre. First, it copies the collection of files in the workspace necessary to translate the specification. This ensures the workspace files are not modified when transformations are applied. Next, it identifies the entry point of the analysis. This is typically a specification but could also be a pattern or type definition, depending on the analysis currently being performed by the user. Finally, it contains a map that tracks all of the renaming performed by the translations. This is necessary to map the signals in the translated SpeAR specification back to the names found in the original specification.

The code that defines the document structure is found at <https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/intermediate/Document.java>

## 7. Transformations

SpeAR transformations accept a SpeAR specification and produce a different, but equivalent, SpeAR specification to make translation to Lustre easier. Transformations must be applied in a specific order due to specific preconditions and postconditions necessary to run each. For example, several transformations rely on operators to be normalized, thus it is essential that the NormalizeOperators transformation be run prior to any transformations that deal with operators. Similarly, it is necessary to expand *in* expressions prior to generating UFC obligations.

This section presents each transformation in its own subsection, in the order it is performed. Each subsection clearly identifies the transformation's preconditions and postconditions so the reader can clearly understand the necessary ordering of the translations.

The code that defines the order of each transformation, and calls each, is at <https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/PerformTransforms.java>

### 7.1. Replacing Abstract Types

*Preconditions:*

- A valid SpeAR document

*Postconditions:*

- A valid SpeAR document with all abstract types replaced with integer types

This transformation replaces any instances of abstract type definitions to named integer type definitions. Since abstract types do not have a structure, this allows models containing them to be analyzed. The reader should note that if an assumption, requirement, or property reasons over an abstract typed value, the analysis will be performed using integer values for the abstract value. Further, type checking only allows abstract typed values to be compared using equality (and inequality).

The code for this transformation is at <https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/ReplaceAbstractTypes.java>

## 7.2. Propagate Predicates

*Preconditions:*

- A valid SpeAR document

*Postconditions:*

- A valid SpeAR document with all predicate subtypes replaced with base types and predicates

Predicate subtypes are refinements of primitive types using predicates. For example, consider a type that contains all the positive integers. This could be represented with the predicate subtype found in Figure 3.

```
posint : {t : int | t > 0}
```

Figure 3 - A predicate subtype defining positive integers

Subtyping in this manner allows the user to compactly and precisely define the types of inputs and outputs a specification will use. However, predicate subtypes are not supported natively by JKind. Thus, it is necessary to translate them such that the semantics of the underlying Lustre representation are equivalent to the SpeAR semantics of a predicate subtype. How this is handled depends whether input or computed variables (outputs or state) are typed by predicate subtypes. The Propagate Predicates transformation ensures this is handled correctly.

For input variables typed by a predicate subtype, this transformation emits an assertion that ensures the input variable will only take on values that satisfy the predicate subtype. Variables that are defined or assigned by the SpeAR specification must be shown to satisfy the predicate subtype. The transformation emits a proof obligation to check that the associated values always satisfy the predicate. If this proof obligation is violated, the user is assigning those variables in ways that violate the predicate subtype's definition.

The code for this transformation is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/PropagatePredicates.java>

## 7.3. Creation of User Namespace

*Preconditions:*

- A valid SpeAR document that contains only user defined variables (if machine-generated variables exist at this point in the they will be incorrectly treated as user-defined variables)

*Postconditions:*

- A valid SpeAR document with all user names prefixed with USER\_

This transformation adds a prefix of USER\_ to every user-named variable in the model. The concept of this translation is that if this USER\_ prefix is added, conflicts with reserved identifiers in the Lustre language are prevented without actually limiting the user's selection for variable names. For example, if the user decides to name a variable "node," which is a reserved keyword in Lustre, the identifier will be replaced with "USER\_node" in the Lustre specification.

The biggest drawback with this approach is a map must be created so names can be mapped back to their original names for presenting analysis results. This map is created as part of the transformation and fed back to the entire analysis process for results presentation.

The code for this transformation is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/CreateUserNamespace.java>

## 7.4. Remove Composite References

*Preconditions:*

- A valid *SpeAR* document

*Postconditions:*

- A valid *SpeAR* document that contains array indexes or record field accesses only in macro variables

This transformation takes all composite type references (array indexing and record field accesses) and creates macros that capture them. The original composite references are replaced with references to the newly created macro. Figure 4 shows a specification before and after running this transformation.

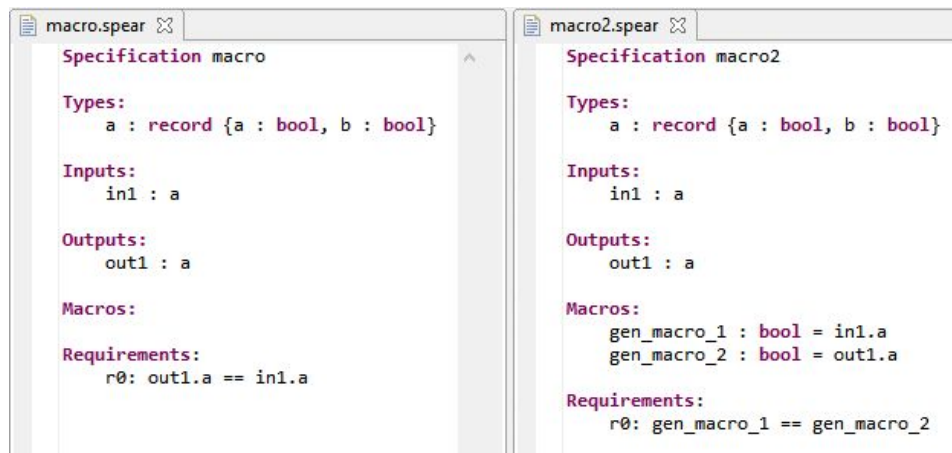


Figure 4 – Before (left) and after (right) running *Remove Composite References*

This transformation is applied so users can equate composite type variable references to specification calls without handling it as a special case in the Lustre translation.

The code for this transformation is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/RemoveCompositeReferences.java>

## 7.5. Replace Shorthand Records

*Preconditions:*

- A valid *SpeAR* document

*Postconditions:*

- A valid SpeAR document that replaces all shorthand record usages with standard record expressions.

This transformation provides the back-end support for a grammatical feature that allows records to be defined in shorthand; that is, the user can supply record field assignments without identifying the field names, provided they are provided in the field order found in the type definition. This transformation replaces shorthand records with traditional records so they can be processed correctly in the translation to Lustre. Table 2 shows an example record type, a shorthand variant of that record type as an expression, and finally, the traditional record resulting from executing this transformation.

<b>Record Type</b>	<b>Shorthand Variant</b>	<b>Post Transformation</b>
r1 : record { a : bool, b : int, c : real }	new r1 {true, 5, 3.1415}	new r1 {a = true, b = 5, c = 3.1415}

Table 2 - Example of a shorthand record

The code for this transformation is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/ReplaceShortHandRecords.java>

## 7.6. Normalize Operators

*Preconditions:*

- A valid SpeAR document

*Postconditions:*

- A valid SpeAR document that uses only one instance of each language operator. For example, “equal to” is replaced with “==”

SpeAR supports multiple options for many operators in the language. For example, the equality operator can be represented with both the “==” operator and an English language equivalent “equal to”. The translation to Lustre is significantly simplified if only one option for each operator is present in the model. Thus, this transformation replaces all operators with their arithmetic operator equivalent.

The code for this transformation is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/NormalizeOperators.java>

## 7.7. Removing Syntactic Sugar

*Preconditions:*

- A valid SpeAR document

*Postconditions:*

- A valid SpeAR document that contains only the base set of PLTL operators; once, historically, since, and triggers

SpeAR features several temporal operators that are found to be useful for SpeAR users. These operators include after/until, before, never, while, and if/then expressions. This transformation replaces those temporal operations with equivalent PLTL expressions. This allows the tool to emit a minimal set of PLTL

expressions during translation and still be able to represent all of SpeAR’s temporal expressions. Table 3 shows the mapping for each temporal operator that is replaced by this transformation.

<u>Temporal Expression</u>	<u>Equivalent Reduced Expression</u>
after P	once P
after P until Q	P triggers not Q
before P	not once P
never P	historically not p
while P then Q	P implies Q
if P then Q	P implies Q

Table 3 - Mapping of temporal expressions to PLTL equivalents

*Note: The reader should make note that the version of triggers used in SpeAR’s translation is slightly different than the academic version of triggers defined in Bounded Verification of Past LTL\* [10]. The version SpeAR uses modifies the initial conditions of the operator as shown in Figure 5 to make the semantics more suitable for SpeAR’s use.*

<pre>node triggers(   a : bool;   b : bool ) returns (   holds : bool ); let   holds = (b and (a or (<b>true</b> -&gt; (pre holds)))); tel;</pre>	<pre>node triggers(   a : bool;   b : bool ) returns (   holds : bool ); let   holds = (b and (a or (<b>false</b> -&gt; (pre holds)))); tel;</pre>
---	--

Figure 5 – Academic (left) and SpeAR (right) version of PLTL **triggers** operator

The code for this transformation is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/RemoveSugar.java>

## 7.8. Expanding “In” Expressions

*Preconditions:*

- A valid SpeAR document

*Postconditions:*

- A valid SpeAR document with all “in” expressions rewritten as conjunctions of relational operators. See below for more details

SpeAR provides an “in” expression that acts as a rudimentary set membership operator. It is able to check if a particular element appears in a constant set, i.e. {2,5,7}, an open or closed interval, or in an array. Lustre does not support this type of operation natively, so “in” expressions must be expanded to create an equivalent (but often less concise) expression. When the right-hand side of the “in” expression is a set or array, this transformation creates a disjunction of equality checks between the left-hand side of the “in” expression and every element possible in the right-hand side of the “in” expression. If the right-hand side of the “in” expression is an interval, the expression is expanded into a conjunction of

relational expressions that capture the interval limits. Table 4 shows a before and after representation of the expansion of several concrete “in” expressions.

<u>Before</u>	<u>After</u>
X in {2,3,9}	X == 2 or X == 3 or X == 9
X in [2,3,9] note: [2,3,9] is an array literal	X == 2 or X == 3 or X == 9
X in Y note: Y is an array of length 4	X == Y[0] or X == Y[1] or X == Y[2] or X == Y[3]
X in {2 ..< 5}	X >= 2 and X < 5

Table 4 - Examples showing the process of expanding “in” expressions

The code for this transformation is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/ExpandInExpressions.java>

## 7.9. Generating UFC Obligations

*Preconditions:*

- A valid SpeAR document

*Postconditions:*

- A valid SpeAR document containing additional observer properties designed for Unique First Cause (UFC) test cases that cover the selected requirements

SpeAR allows users to generate tests for requirements or properties that achieve Unique First Cause [11] (UFC) coverage. The user can enable this feature by adding the *ufc* flag to the constraint of interest, similar to the *observe* flag. This pass generates observer properties for each logical clause in the constraint, which are then translated to Lustre and passed to JKind to find counterexamples. These counterexamples are the test cases that validate that each clause in the requirement contributes to the evaluation of the requirement.

The generation of UFC test cases is accomplished by examining the logical structure of the code and emitting *trap properties*. Trap properties are properties designed to be false and whose resultant counterexample contains the test case of interest. UFC is a requirements coverage metric. It is concerned with demonstrating that each condition in a logical formula contributes to the evaluation of the requirement. Consider a simple requirement ***r1*: a or b** on a system with two inputs, ***a*** and ***b***. One way to rigorously test ***r1*** is to develop two test cases; one that shows ***r1*** is true when ***a*** is true and ***b*** is false and a second that shows ***r1*** is true when ***b*** is true and ***a*** is false. The instances when ***r1*** are false are not considered; requirements should always hold and thus it is not necessary to test those instances.

The UFC test generation capability will emit test cases for logical expressions found in SpeAR requirements. Expressions that the UFC test generator will explore are:

- and, or, xor, implies, not
- previous (with initial value)
- if then else

All other expressions are treated as atomic and will not be further decomposed.

The code for this transformation is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/GenerateUFCObligations.java>

## 7.10. Replace Specification Calls

*Preconditions:*

- *A valid SpeAR document*

*Postconditions:*

- *A valid SpeAR document containing only normalized specification calls*

SpeAR puts limitations on the way that specification calls can be used. For example, they may not appear within macro definitions. Secondly, the results of a specification call must be equated with local or output variables within an existing specification. Due to these limitations, and to simplify translation to Lustre, specification calls are normalized; that is, the call and the variables assigned are combined into a single grammatical element.

The code for this translation is at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/ReplaceSpecificationCalls.java>

## 7.11. Uniquify Normalized Calls

*Preconditions:*

- *A valid SpeAR document containing only normalized specification calls*

*Postconditions:*

- *A valid SpeAR document with each normalized specification call mapped to exactly one specification*

SpeAR allows users to call a node multiple times. This introduces complexity into the Lustre translation because each individual node call potentially has state information. One way to simplify the tracking of the state is to copy each called node and give it a unique name. Then, the mapping between node calls and the state information for each call is one-to-one. This transformation performs this simplification on the Normalized Calls resulting from the previous pass.

The code for this transformation can be found at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/transformations/UniquifyNormalizedCalls.java>

## 8. Translation to Lustre

SpeAR is designed to support constraint-based analyses. This means that a SpeAR specification can have multiple satisfying implementations. Modeling this behavior in Lustre is not trivial; a typical Lustre model will have exactly one interpretation. To capture SpeAR's behavior correctly, it is necessary to model both SpeAR specification inputs and outputs as Lustre inputs. This allows JKind to assign both inputs and outputs to any possible value for the associated type. SpeAR requirements are emitted as constraints between the input and output values. These constraints limit the possible assignments between inputs and outputs, ensuring that assignments satisfy requirements. Shown below is an example of a simple SpeAR specification and its corresponding translation into Lustre.



<b>Specification</b> simple  <b>Inputs:</b> $a : \text{int}$  <b>Outputs:</b> $x : \text{int}$  <b>Requirements:</b> $r0: x > a$  <b>Properties:</b> $p0: x - a > 0$	<pre> node USER_simple(   USER_a : int;   USER_x : int ) returns (   constraints : bool ); var   USER_r0 : bool;   USER_p0 : bool; let   --%MAIN;    USER_r0 = (USER_x &gt; USER_a);   USER_p0 = (constraints =&gt; ((USER_x - USER_a) &gt; 0));   constraints = historically(USER_r0);   --%PROPERTY USER_p0; tel; </pre>
--	--

Figure 6 - Example SpeAR specification and its Lustre representation

In this translation, each input **USER\_a** and **USER\_x** (which correspond to  $a$  and  $x$  in the original SpeAR) can be assigned to any integer. Since SpeAR and Lustre support infinite range integers, this means that both can be assigned any value in  $(-\infty, \infty)$  in the analysis. However,  $x$  has a constraint (a requirement) **r0** on it that requires it to be greater than  $a$ , hence it can only be assigned values from  $(a, \infty)$ . This constraint is captured as **USER\_r0** in the Lustre representation. Finally, the property **p0** checks that when all of the model's constraints hold (in this case, just **r0**) for every step (*historically*) then  $x - a > 0$ . This property should hold if  $x$  is always greater than  $a$ , which is specified as requirement **r0**. Analysis of the Lustre file by JKind returns the expected result: property p0 holds.

The following subsections describe the internal representation and how the major elements of that representation are translated to Lustre.

### 8.1. Internal Representation

The translator creates an intermediate representation after applying the transformations described in Section 7 to the document. The intermediate representation defines the structure of the final Lustre representation without actually performing the translation; this allows an opportunity to gather and establish unique names for referenced types, constants, patterns, and specifications. After establishing the structure of the translation in the intermediate representation, targeting Lustre is straightforward.

The code for the internal representation can be found at

<https://github.com/lgwagner/SpeAR/tree/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master>

Careful inspection of this code shows that the top-level element of the translation is found within the SProgram class. From there, SpeAR elements are parsed into corresponding elements of the representation. For example, specifications are translated into the SSpecification class, type definitions into the STypeDef class, constants into the SConstant class, and so on. Each of these classes utilize the naming map of the Lustre element they will reside within to assign a unique name. Each class also defines how to generate the corresponding Lustre constructs for it. Once the class hierarchy is defined, the translation to Lustre is accomplished by walking the intermediate representation and invoking the methods that produce Lustre for each element.

The following link highlights a few lines of code that provide an example of methods that are used to emit equivalent Lustre.

<https://github.com/lgwagner/Spear/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master/SMacro.java#L43-L49>

In this example, the `toVarDecl` method creates a Lustre variable declaration for the Macro object while the `toEquation` method creates a Lustre equation. Both methods accept an `SSpecification` as an argument. This is so the translation can access the naming map for the element it will appear within and any other contextual information it might need.

## 8.2. Type and Expression Mapping

SpeAR types map one-to-one to Lustre types with one exception. Predicate subtypes map to their base types in Lustre. This, combined with the `PropagatePredicates` transformation identified in Section 7.2, ensures the behavior of predicate subtypes in Lustre matches the semantics in SpeAR.

Many expressions in SpeAR map seamlessly to expressions in Lustre. For example, all arithmetic, relational, and logical operators map directly to existing Lustre expressions. Some expressions do not have a direct mapping into Lustre. This includes elements like temporal expressions (discussed in Section 7.7), previous expressions, bit to integer (`btoi`) casting, floating point modulus (`fmod`), and the count and consecutive count (`ccount`) operators. Table 5 identifies how irregular SpeAR expressions are translated to Lustre. Note that the temporal operations encountered at this stage of the translation are limited as a result of performing the `Remove Syntactic Sugar` transformation described in Section 7.7.

SpeAR expression	Equivalent Lustre expression
once p	<code>once_p = p or (false -&gt; pre once_p)</code>
historically p	<code>historically_p = p and (true -&gt; pre historically_p)</code>
p triggers q	<code>p_triggers_q = b and (a or (false -&gt; p_triggers_q))</code>
p since q	<code>p_since_q = b or (a and (false -&gt; p_since_q))</code>
previous p	<code>previous_p = pre p</code>
previous p with initial value q	<code>previous_p = (q -&gt; pre p)</code>
btoi p	<code>btoi p = if p then 1 else 0</code>
a fmod b	<code>a_fmod_b = (a - (b * real(floor((a/b)))))</code>
count p	<code>count_p = if p then 1 else 0 + (0 -&gt; pre count_p)</code>
ccount p	<code>ccount_p = if p then (1 -&gt; pre ccount_p) else 0</code>

Table 5 - Mapping of irregular SpeAR expressions into Lustre

## 8.3. Translation of Normalized Specification Calls

SpeAR allows specifications to call to other specifications to assign variables. This allows users to modularize and reuse portions of specifications. Each specification call and variables it is assigning are combined into a Normalized Call, discussed in 7.10. Figure 7 shows a specification, *top*, that calls a second specification named *called*. The call to specification *called*, found in requirement *r0* of specification *top*, assigns variable *x* in the *top* specification. These are combined into a normalized call that contains both variable *x* and the call to specification *called*.

<pre> Specification top  Imports:   import "called.spear"  Inputs:   a : int  Outputs:   x : int  Assumptions:   a0: a &gt; 0  Requirements:   r0: x == spec called(a)  Properties:   p0: x &gt; 1 </pre>	<pre> Specification called  Inputs:   a : int  Outputs:   x : int  Assumptions:   a0: a &gt; 1  Requirements:   r0: x &gt; a </pre>
---	---

Figure 7 - Simple specification that calls another specification

Figure 8 shows the translated Lustre for the specifications shown in Figure 7. As previously discussed, the inputs and outputs of a SpeAR specification are represented as inputs in the Lustre representation. When a specification is called, it must be translated so the specification inputs *and outputs* are provided as input arguments to the call. Further, no output variable is returned back to the calling specification. Instead, the constraints applied in the called specification are conjuncted and returned to the caller in the form of a single boolean variable. This is so the caller can add the called specification's constraints into its own constraints. Careful inspection of Figure 8 shows that the call to USER\_called provides input *a* and output *x* as input arguments. Further, the constraints of USER\_called are captured and provided back to USER\_top within the constraints variable. These constraints are fed back into USER\_top's constraints for the system.

<pre> node USER_top(   USER_a : int;   USER_x : int ) returns (   constraints : bool ); var   USER_a0 : bool;   USER_r0 : bool;   USER_p0 : bool; let   --%MAIN;   USER_a0 = (USER_a &gt; 0);    USER_r0 = USER_called(USER_a, USER_x);   USER_p0 = (constraints ==&gt; (USER_x &gt; 1));   constraints = historically(USER_r0);   assert USER_a0;   --%PROPERTY USER_p0; tel; </pre>	<pre> node USER_called(   USER_a : int;   USER_x : int ) returns (   constraints : bool ); var   USER_a0 : bool;   USER_r0 : bool; let   USER_a0 = (USER_a &gt; 1);   USER_r0 = (USER_x &gt; USER_a);   constraints = USER_r0; tel; </pre>
---	--

Figure 8 - Lustre translations for logical entailment analysis of the specifications from Figure 7

This treatment of specification variables allows the tool to maintain a constraint-based approach to requirements analysis in Lustre while still allowing specification calls.

Calls from one specification to another are resolved in a two-step process. First, the calls from the current specification being processed are gathered. Here we introduce variables that correspond to the

inputs and outputs of the called specification. These inputs and outputs are captured and noted for each call so that they can easily be identified on the second translation step, which is directly translating SpeAR to Lustre. At that time, calls are linked with the newly introduced variables that correspond to the called specifications inputs and outputs.

The first step of the translation that aggregates all of the calls within a specification and creates fresh variables in the calling specification is shown at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master/SProgram.java#L75-L76>.

The second step of the translation that translates the calling specification to Lustre and feeds the correct inputs and outputs to the called specification is shown at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/lustre/TranslateExpr.java#L78-L92>.

#### 8.4. Observer Properties

Observer properties are properties for which the user expects the tool to provide a trace that exhibits the condition of interest. Observers can be used to validate that a specification behaves as intended by providing a concrete trace that satisfies the requirements. Observers can also be used to provide the user with a requirements-based test case.

SpeAR observers are simply properties that are flagged with the ‘observe’ flag. These properties are negated during translation. JKind attempts to prove the condition of interest never occurs. If it can occur, it provides back a counterexample that demonstrates how. This counterexample observes, or provides a witness to, the property of interest.

#### 8.5. Commonality

SpeAR can analyze a specification and show that 1) its requirements logically entail its properties (that it behaves or does not behave in a certain way), 2) it is logically consistent, and 3) it is realizable. Each of these analyses result in similar Lustre translations with slight variances. For instance, the types, constants, inputs, outputs, local variables, and called specifications remain the same. However, properties emitted for logical entailment analysis do not appear in the Lustre files for logical consistency or realizability analysis.

The function that translates SpeAR to base Lustre are found in

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master/SSpecification.java#L132-L174>.

Logical entailment is translated using two different functions. The first function translates the top level SpeAR specification for analysis. Essentially this function takes the base Lustre translation and adds an assertion for all of the assumptions, and properties for the SpeAR properties, and selects the requirements of the specification as IVC principals for model checking analysis. This function is found at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master/SSpecification.java#L176-L196>.

The second function translates called nodes for logical entailment by adding an assertion equation for the called assertions to the base Lustre. This code is found at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master/SSpecification.java#L211-L215>.

The translation for logical consistency is similar to the translation for logical entailment, except it does not check properties as they have no impact on the consistency of the specification. Logical consistency uses the base Lustre translation but adds a signal that attempts to prove that requirements cannot be satisfied for N steps, which is identified by user in the settings. This code is found at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master/SSpecification.java#L217-L233>.

Logical consistency called nodes are translated the same as logical entailment called nodes and this is reflected in the code at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master/SSpecification.java#L249-L251>.

The logical consistency translation for the example in Figure 7 is shown in Figure 9.

```
node USER_top(  
  USER_a : int;  
  USER_x : int  
) returns (  
  constraints : bool  
)  
;  
var  
  USER_a0 : bool;  
  USER_r0 : bool;  
  USER_p0 : bool;  
  counter : int;  
  consistent : bool;  
let  
  --%MAIN;  
  counter = (1 -> ((pre counter) + 1));  
  
  USER_a0 = (USER_a > 0);  
  
  USER_r0 = USER_called1(USER_a, USER_x);  
  
  USER_p0 = (constraints => (USER_x > 1));  
  
  consistent = (not (constraints and (counter >= 10)));  
  
  constraints = historically((USER_a0 and USER_r0));  
  
  --%PROPERTY consistent;  
  
  --%IVC USER_a0, USER_r0;  
  
tel;  
  
node USER_called(  
  USER_a : int;  
  USER_x : int  
) returns (  
  constraints : bool  
)  
;  
var  
  USER_a0 : bool;  
  USER_r0 : bool;  
  counter : int;  
let  
  counter = (1 -> ((pre counter) + 1));  
  
  USER_a0 = (USER_a > 1);  
  
  USER_r0 = (USER_x > USER_a);  
  
  constraints = USER_r0;  
  
tel;
```

Figure 9 – Logical consistency translation for the example in Figure 7

Finally, the translation for realizability is similar to the translation for logical consistency, except the realizability analysis tool requires the true inputs of the specification to be identified. This code is found at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master/SSpecification.java#L235-L247>.

In realizability analysis, called specifications are translated the same as called specifications are for logical entailment, except that the specification inputs are marked as inputs for the realizability tool. Similar to the logical entailment and logical consistency analysis, specification assumptions are translated as Lustre assertions. The jRealizability tool keys on Lustre assertions for its contract realizability analysis.

The realizability translation for the example in Figure 7 is shown in Figure 10.

```
node USER_top(  
  USER_a : int;  
  USER_x : int  
) returns (  
  constraints : bool  
);  
var  
  USER_a0 : bool;  
  USER_r0 : bool;  
  USER_p0 : bool;  
  counter : int;  
let  
  --%MAIN;  
  counter = (1 -> ((pre counter) + 1));  
  
  USER_a0 = (USER_a > 0);  
  
  USER_r0 = USER_called(USER_a, USER_x);  
  
  USER_p0 = (constraints => (USER_x > 1));  
  
  constraints = historically(USER_r0);  
  
  assert USER_a0;  
  
  --%PROPERTY constraints;  
  
  --%REALIZABLE USER_a;  
  
tel;  
  
node USER_called(  
  USER_a : int;  
  USER_x : int  
) returns (  
  constraints : bool  
);  
var  
  USER_a0 : bool;  
  USER_r0 : bool;  
  counter : int;  
let  
  counter = (1 -> ((pre counter) + 1));  
  
  USER_a0 = (USER_a > 1);  
  
  USER_r0 = (USER_x > USER_a);  
  
  constraints = USER_r0;  
  
tel;
```

Figure 10 - Realizability translation for the example in Figure 7

## 8.6. Patterns Analysis Translation

The translation of the pattern analysis is similar to the process for specifications, as discussed in Section 8.1. When a pattern is the entry point for analysis, a document is created that includes all of the imported files, constants, and types necessary to execute the pattern. Then, all of dependencies to analyze the pattern are captured in the SProgram class. Finally, the SPattern class captures the inputs, outputs, locals, equations, assertions, and properties of the original pattern. Once in this form, translation to Lustre is accomplished by translating the SProgram elements, including the main SPattern object, into Lustre.



The code for handling the translation of Patterns can be found at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master/SProgram.java#L82-L103>

and

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.translate/src/com/rockwellcollins/spear/translate/master/SPattern.java>.

<pre> pattern add_one(input : int) returns (output : int) var   p1 : bool let   output = input + 1    p1 = output &gt; input   --%PROPERTY p1 tel </pre>	<pre> node add_one(input : int) returns (output : int); var   p1 : bool; let   output = input + 1;    p1 = output &gt; input;   --%PROPERTY p1; tel; </pre>
--	---

Figure 11 - A SpeAR pattern and it's corresponding Lustre translation

Figure 11 shows a simple SpeAR pattern that accepts an integer as inputs and returns an integer as output that is one greater than the input. It also contains a property that checks if the output is greater than the input. Figure 11 also shows the corresponding Lustre representation.

## 9. Results Presentation

SpeAR's analyses are performed by JKind. The results of these analyses must be interpreted in a way that is consistent with the context of the analysis the user is running in SpeAR. For example, if a user is trying to prove a property in Logical Entailment, and it fails, SpeAR should identify that something undesired happened. This situation is handled by displaying a red box with an exclamation mark near to the property, as shown in Figure 12, for property p0.


































<b>Properties:</b> <p>p0: mode == COOKING <b>implies not</b> door_closed</p> <p>p1: before start <b>implies</b> mode == SETUP</p> <p>p2: clear <b>implies</b> mode &lt;&gt; COOKING</p> <p>p3: cook_time == 0 <b>implies</b> mode &lt;&gt; COOKING</p> <p>//design an observer to show a trace in  //which the microwave gets into the cooking mode</p> <p>obs1 <b>observe:</b> mode == COOKING</p> <p>//design an observer to show a trace in which the  //microwave gets into the cooking mode for 3 consecutive steps</p> <p>obs2 <b>observe:</b> ccount(mode == COOKING) == 6</p> <p>//design an observer to show a trace in which the  //microwave goes from suspended to cooking</p> <p>obs3 <b>observe:</b> (pre_mode == SUSPENDED) <b>and</b> (mode == COOKING)</p> <p>//trivial conflict</p> <p>obs4 <b>observe:</b> mode == COOKING <b>and</b> mode == SUSPENDED</p>																									
<b>Analysis Results</b> <table border="1"> <thead> <tr> <th>Property</th><th>Result</th></tr> </thead> <tbody> <tr> <td> p0</td><td>Invalid (0s)</td></tr> <tr> <td> p1</td><td>Valid (1s)</td></tr> <tr> <td> p2</td><td>Valid (1s)</td></tr> <tr> <td> p3</td><td>Valid (2s)</td></tr> <tr> <td> obs1</td><td>Valid (0s)</td></tr> <tr> <td> obs2</td><td>Valid (1s)</td></tr> <tr> <td> obs3</td><td>Valid (1s)</td></tr> <tr> <td> obs4</td><td>Invalid (1s)</td></tr> <tr> <td> obs5</td><td>Valid (1s)</td></tr> <tr> <td> USER_pre_cook_time_satisfies_predicate</td><td>Valid (1s)</td></tr> <tr> <td> USER_cook_time_satisfies_predicate</td><td>Valid (1s)</td></tr> </tbody> </table>		Property	Result	 p0	Invalid (0s)	 p1	Valid (1s)	 p2	Valid (1s)	 p3	Valid (2s)	 obs1	Valid (0s)	 obs2	Valid (1s)	 obs3	Valid (1s)	 obs4	Invalid (1s)	 obs5	Valid (1s)	 USER_pre_cook_time_satisfies_predicate	Valid (1s)	 USER_cook_time_satisfies_predicate	Valid (1s)
Property	Result																								
 p0	Invalid (0s)																								
 p1	Valid (1s)																								
 p2	Valid (1s)																								
 p3	Valid (2s)																								
 obs1	Valid (0s)																								
 obs2	Valid (1s)																								
 obs3	Valid (1s)																								
 obs4	Invalid (1s)																								
 obs5	Valid (1s)																								
 USER_pre_cook_time_satisfies_predicate	Valid (1s)																								
 USER_cook_time_satisfies_predicate	Valid (1s)																								

Figure 12 - Analysis results for a sample specification

However, if a user is analyzing an observer property, SpeAR is constructing a property that it expects to fail with an accompanying counterexample. In that situation, the tool should indicate to the user that something desirable has happened. For observer obs1, in Figure 12, this appears as a spreadsheet icon

to inform the user that the result was a trace, which is expected by the user. JKind does not have any intuition on whether the property being verified is expected to be true or false; SpeAR presents results back to the user such that they make sense in the context of the analysis intended by the user.

The remainder of this section addresses the formatting and presentation of analysis results for each of SpeAR's formal methods analyses.

### 9.1. Logical Entailment Analysis Results

Logical entailment analysis checks properties of two types: properties of correctness and observer properties. Correctness properties attempt to demonstrate that a property holds, for every possible input and state combination, on the given set of requirements and assumptions. When the tool finds a trace that violates a property, it must be returned back to the user so that it can be fixed. Valid correctness properties are represented with a green check box, as shown in Figure 12 for property p1.

Observer properties are designed to find a single trace that observes the condition of interest. If a negated observer property is proven, then there are no traces allowed by the requirements that observe the condition of interest. SpeAR represents this with a red exclamation point (similar to a violated correctness property), as shown in Figure 12 for observer property obs4. When a trace satisfying the observer is found, the result is shown with a miniature spreadsheet icon, as shown for obs1 in Figure 12.

The code for displaying the Logical Entailment analysis results is found at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.ui/src/com/rockwellcollins/spear/ui/views/SpearEntailmentResultsView.java>

and

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.ui/src/com/rockwellcollins/spear/ui/views/SpearEntailmentMenuListener.java>

### 9.2. Traceability

Logical entailment analysis can also generate a traceability matrix that identifies which requirements are needed to prove a given property. This feature is enabled in the SpeAR preferences menu by checking the "Enable IVC during Logical Entailment Analysis" option. Once enabled the tool will provide an option to generate a traceability matrix that identifies which requirements prove the given properties. Figure 12 shows just the properties section of a SpeAR specification for a simple microwave oven.



```

Inputs:
  start : bool
  clear : bool
  door_closed : bool
  user_time : posint

Outputs:
  mode : mode_type

State:
  cook_time : posint

Macros:
  pre_cook_time : posint = previous cook_time with initial value 0
  pre_mode : mode_type = previous mode with initial value SETUP

Assumptions:
  //start and clear will not be pressed simultaneously
  a0: not (start and clear)

Requirements:
  //mode is defined by the state machine implemented in pattern "mode_machine"
  r0: mode == mode_machine(start,clear,door_closed,user_time,cook_time)

  r1: if mode == SETUP
    then cook_time == 0
    else if (pre_mode == SETUP) and (mode <> SETUP)
    then cook_time == user_time
    else if (pre_mode == COOKING) and (mode == COOKING)
    then cook_time == pre_cook_time - 1
    else cook_time == pre_cook_time

```

Figure 13 - The Inputs, Outputs, State, Macros, Assumptions, and Requirements for the Specification shown in Figure 12.

Figure 14 shows the traceability matrix generated for the specification in Figure 13 and Figure 12.

Traceability Matrix		
	r0	r1
p1	X	
p2	X	
p3	X	X
USER_pre_cook_time_satisfies_predicate	X	X
USER_cook_time_satisfies_predicate	X	X

Figure 14 - Traceability matrix

This example shows that all properties depend on requirement r0. This makes sense because requirement r0 specifies the mode transition behavior of the entire system being described. Properties p3 and the predicates for the cook\_time and pre\_cook\_time variables depend on requirement r1 as well.

This analysis can be used to show requirements coverage by a given set of properties or vice-versa. For example, if a given set of properties do not cover all requirements, the user must determine if more properties are required to cover the requirement or if some requirements are unnecessary. Regardless, traceability can identify gaps in requirements or property coverage.

Traceability information is obtained by augmenting the logical entailment translation by identifying the requirements in the specification as IVC principal signals. For the example in Figure 13 and Figure 12, this is accomplished by adding the following line at the end of the main Lustre node:

```
--%IVC USER_a0, USER_user_time_satisfies_predicate, USER_r0, USER_r1;
```

This tells the tool that when IVC is enabled (it is a user configurable option) it should use assumptions **a0**, **USER\_user\_time\_satisfies\_predicate** (from the predicate subtype) and requirements **r0** and **r1** as IVC principals. When a property is proven, IVC will identify which subset of {**USER\_a0**, **USER\_user\_time\_satisfies\_predicate**, **USER\_r0**, **USER\_r1**} were necessary to prove it. This information is used to generate the traceability matrix in Figure 14.

The traceability matrix is drawn using standard Java drawing utilities. The code for drawing the traceability matrix is shown in

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.ui/src/com/rockwellcollins/spear/ui/views/SpearTraceabilityMatrixView.java>.

### 9.3. Logical Consistency

Logical consistency seeks to identify one trace that satisfies the user's requirements to a depth N, which is configurable in the SpeAR preferences menu. The purpose of this check is to ensure that the given set of requirements are free from a major conflict that would prevent them from having any satisfying traces. If the logical consistency property has at least one satisfying trace, the result is presented with a green check box, as shown in Figure 15. Right-clicking on the result gives the user the option to view the satisfying trace, which is the counterexample provided by JKind.

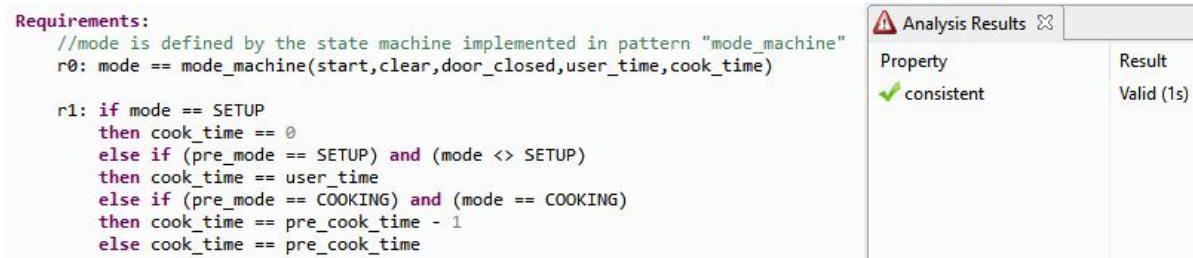


Figure 15 - Logical Consistency analysis results

If the logical consistency property does not have any satisfying traces, it is represented with a red exclamation point (not shown).

The code for displaying the logical consistency results is found at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.ui/src/com/rockwellcollins/spear/ui/views/SpearConsistencyResultsView.java>

and

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.ui/src/com/rockwellcollins/spear/ui/views/SpearConsistencyMenuListener.java>

## 9.4. Realizability

Realizability analysis attempts to show that all traces in a given model are free from conflict. It is a much more rigorous check than the logical consistency check but also scales very poorly to large models. If a model has realizability conflicts, SpeAR represents it with a red exclamation point, as shown in Figure 16. Right-clicking on the result allows the user to view the counterexample provided by JKind.




<b>Specification</b> realizability	<b>Analysis Results</b>				
<b>Inputs:</b> a : int	<table><thead><tr><th>Property</th><th>Result</th></tr></thead><tbody><tr><td> result</td><td>Invalid (0s)</td></tr></tbody></table>	Property	Result	 result	Invalid (0s)
Property	Result				
 result	Invalid (0s)				
<b>Outputs:</b> b : int					
<b>Requirements:</b> r1: a==3 implies b==2 r2: a==3 implies b==3					
<b>Properties:</b> p0: not (a==3)					

Figure 16 - Realizability analysis results

If all traces in a given set of requirements are free from conflict, SpeAR will provide a green checkmark (not shown). Finally, for large models the analysis may timeout and provide back a yellow exclamation point (not shown), which suggests no conflicts were found but the analysis was not complete. Unknown is returned for models with nonlinearity as the underlying solver cannot handle nonlinear models.

The code for presenting realizability analysis results can be found at

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.ui/src/com/rockwellcollins/spear/ui/views/SpearRealizabilityResultsView.java>

and

<https://github.com/lgwagner/SpeAR/blob/master/com.rockwellcollins.spear.ui/src/com/rockwellcollins/spear/ui/views/SpearRealizabilityMenuListener.java>.

## 10. References

- [1] A. Ficarek, L. Wagner, E. Hoffman, B. Rodes, M. A. Aiello and J. Davis, "SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements," in *NASA Formal Methods*, 2017.
- [2] Oracle, "Java webpage," [Online]. Available: [java.oracle.com](http://java.oracle.com).
- [3] Eclipse Foundation, "Eclipse Integrated Development Environment Homepage," [Online]. Available: <http://www.eclipse.org>.
- [4] Eclipse Foundation, "Xtext language development framework," [Online]. Available: <https://www.eclipse.org/Xtext/>.
- [5] Wikimedia, "Lustre Wikipedia page," [Online]. Available: [https://en.wikipedia.org/wiki/Lustre\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Lustre_(programming_language)).
- [6] A. Gacek, "JKind source repository," [Online]. Available: <https://github.com/agacek/jkind>.
- [7] A. Gacek, A. Katis, M. Whalen, J. Backes and D. Cofer, "Towards realizability checking of contracts using theories," in *NASA Formal Methods (NFM)*, 2015.
- [8] G. Elaheh, M. Whalen and A. Gacek, "Efficient Generation of All Minimal Inductive Validity Cores," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2011.
- [9] "Xtext Grammar language," [Online]. Available: [https://www.eclipse.org/Xtext/documentation/301\\_grammarlanguage.html](https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html).
- [10] A. Cimatti, M. Roveri and D. Sheridan, "Bounded Verification of Past LTL," in *Formal Methods in Computer-Aided Design*, 2004.
- [11] M. W. Whalen, A. Rajan, M. P. Heimdahl and S. P. Miller, "Coverage Metrics for Requirements-based Testing," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, Portland (Maine), 2006.