Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 116 (2005) 185–198
www.elsevier.com/locate/entcs

A Compositional Framework for Formally Verifying Modular Systems [1]

Carlo A. Furia and Matteo Rossi[2]

Dipartimento di Elettronica e Informazione, Politecnico di Milano
32, Piazza Leonardo da Vinci, 20133 Milano, Italy

Abstract

We present a tool-supported framework for proving that the composition of the behaviors of the separate parts of a complex system ensures a desired global property of the overall system. A compositional inference rule is formally introduced and encoded in the logic of the PVS theorem prover. Methodological considerations on the usage of the inference rule are presented, and the framework is then used to prove a meaningful property of a simple, but significant, control system.

Keywords: Formal verification, modular systems, real-time, compositionality.

1 Introduction

As systems grow in size, being able to subdivide them in components is of crucial importance to keep their complexity under control. In particular, one would like to specify and design single components separately, and then be able to guarantee that they also behave correctly when they interact with each other. In fact, parts that operate properly under some assumptions on the behavior of the external world might misbehave when interacting with other elements of the overall system that do not satisfy those assumptions (for example, a data analyzer that accepts inputs with frequency r might

[1] Work supported by the MIUR project: "QUACK: Piattaforma per la qualità di sistemi embedded integrati di nuova generazione"
[2] Email: rossi@elet.polimi.it

doi:10.1016/j.entcs.2004.02.076

work improperly when connected to a sensor that sends data with frequency $2r$).

Formal methods are more and more recognized to be a useful tool for the development of applications, especially critical ones, as they allow to precisely verify the correctness of systems in their early development phases, before uncaught mistakes become overly costly to fix, or even catastrophic. One problem often attributed to formal methods, however, is that they do not "scale up", i.e. when the system grows in complexity, they are too cumbersome and unwieldy to be used effectively.

A compositional framework can help in this regard, in that it would allow one to focus on the single parts of the system at first, and analyze their mutual interactions at a later moment, with a smaller effort than it would be required if all aspects (local and global) of the application were taken into account at once at integration time. A good, proof-oriented compositional framework must be based on sound inference rules that allow one to deduce global properties of the system from the behavior of its single parts. In addition, for the framework to be actually usable, it should be supported by (semi)automatic tools that facilitate the analysis of the modeled systems.

Rules for composing single specifications into complex systems have been studied in the past [2], also, but not only, with reference to temporal logics [1] [3]. This paper presents a valid inference rule for the TRIO specification language [5,6] that is suitable to formally prove the correctness of the behavior of a modular system from the behavior of its components. The rule has been encoded in the logic of the PVS theorem prover [7], and support strategies have been developed.

The paper is structured as follows: Section 2 shortly introduces the TRIO language, using the specification of the application analyzed in Section 5 as an example; Section 3 presents the inference rule on which our compositional framework for TRIO is based; Section 4 describes the PVS-based tool that supports the framework; Section 5 introduces some methodological considerations on the use of the compositional framework and shows how it can be applied to a simple, but meaningful, control system; Section 6 draws some conclusions and outlines future work in this line of research.

2 TRIO

TRIO [5,6] is a typed linear metric temporal logic enriched with object-oriented and modular features for writing specifications of complex systems.

[3] For the sake of space limit, we do not present extensively the literature related to this research. The interested reader can refer to [3] for a comparison with relevant related works.

Operator	TRIO Definition
$Past(A,d)$	$d > 0 \wedge Dist(A,-d)$
$Futr(A,d)$	$d > 0 \wedge Dist(A,d)$
$Som(F)$	$\exists d\ Dist(F,d)$
$Alw(F)$	$\forall d\ Dist(F,d)$
$AlwP(F)$	$\forall d\ (d > 0 \Rightarrow Past(F,d))$
$AlwP_i(F)$	$AlwP(F) \vee F$
$Lasts(F,t)$	$\forall d\ (0 < d < t \Rightarrow Futr(F,d))$
$Lasted(F,t)$	$\forall d\ (0 < d < t \Rightarrow Past(F,d))$
$UpToNow(F)$	$\exists d\ (d > 0 \wedge Lasted(F,d))$
$NowOn(F)$	$\exists d\ (d > 0 \wedge Lasts(F,d))$

Table 1
Derived Temporal Operators

Each TRIO formula is evaluated with respect to the current time instant, which is left implicit. The basic temporal operator is called *Dist* and relates other instants of time with the current one: $Dist(F,t)$ is true of a time-dependent formula F if and only if F holds at a time instant which is t time units apart from the current one. Combining the *Dist* operator with all common propositional operators and quantifiers of first-order logic, we define a number of *derived* temporal operators, some of which are shown in Table 1.

Notice that TRIO is well suited to deal with both continuous and discrete time; if the temporal domain is discrete, the definition of some temporal operators changes slightly with respect to the one shown in Table 1. The basic elements of a TRIO specification (predicates, functions, etc.) are called *items*. *Events* and *states* are items with a particular temporal behavior (e.g. events are predicates that are true only in isolated instants).

Let us illustrate the features of TRIO by means of a simple, but meaningful example (which will also be used in Section 5 to show an application of our compositional framework). The case study consists of a reservoir and a controller. Whenever the level of liquid in the reservoir is below a certain threshold, the controller opens a valve to fill it. Moreover, the reservoir can nondeterministically leak.

Example 2.1 [Items of the reservoir] The reservoir being filled/leaking is modeled by the TRIO states *filling* and *leaking*, respectively. Conversely, the current level of fluid in the reservoir is modeled by a time-dependent item named *level*.

There are three categories of TRIO formulae: axioms, assumptions and theorems. Axioms and assumptions postulate the basic behavior of the system, while theorems describe properties that can be derived from them.

Example 2.2 [Axioms of the reservoir] The temporal evolution of the level of liquid in the reservoir is modeled by means of four axioms (named level_behavior_1/2/3/4), which take into account all the possible configurations the reservoir can be in: filling and leaking, just filling, just leaking, neither filling nor leaking. If fr and lr are the filling and leaking rate, respectively, axiom level_ behavior_1, that corresponds to the situation in which the reservoir is both being filled and leaking, is the following[4] :

axiom 1 (level_behavior_1)
$Lasted(\text{filling} \wedge \text{leaking}, t) \wedge Past(\text{level} = l, t) \Rightarrow \text{level} = l + (fr - lr) \cdot t$

TRIO is enriched with object-oriented constructs to support inheritance, genericity and modularization. The basic encapsulation unit is the *class*, a collection of items, formulae and modules[5] . The semantics of a composite class is given by the logical conjunction of all the axioms of all the modules.

Example 2.3 [Reservoir and controller parameters] The controller class is parametric with respect to the lower and upper bounds L_l and L_u between which the liquid level must stay. The reservoir class is also parametric with respect to the filling and leaking rates fr and lr. Parameter Δ of the controller class, in addition, defines a delay in the control action: whenever the level of fluid stays below the upper bound L_u for more than Δ time units, the controller issues a filling action till the level grows back to L_u. This is formalized by axiom filling_def of the controller class shown below.

axiom 2 (filling_def) $\text{filling} \Leftrightarrow Lasted(\text{level} < L_u, \Delta)$

Example 2.4 [The reservoir system] The reservoir system analyzed in Section 5 is modeled by the class reservoir_system represented in Figure 1: it is built by composing an instance of a reservoir class, describing a reservoir containing liquid, and a controller class, describing its controller.

The ultimate goal of the analysis performed on the system is to prove that the level of the liquid in the reservoir always stays between the upper and lower bounds L_u and L_l. This is formalized by the following theorem of class reservoir_system:

[4] TRIO formulae are implicitly temporally closed with the Alw operator.
[5] in TRIO terms, a module is an instance of a class.

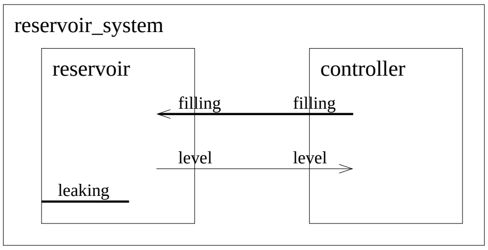

Fig. 1. The reservoir system

theorem 3 (level_stays_between_bounds) $L_l \leq \text{reservoir.level} \leq L_u$

3 A Compositional Inference Rule

This section presents a compositional inference rule for the TRIO language. More precisely, we are considering compositional specifications written in the rely/guarantee paradigm [2,1]. This rule will be used in Section 5 to derive formal properties of the composite system introduced in Section 2. For the sake of brevity, we do not demonstrate the soundness of the inference rule (see [3] for further details).

Let us consider a system composed by n modules $C_1, \dots, C_n$. To each module $i = 1, \dots, n$ we associate an assumption E_i about the behavior of its environment and a behavioral property M_i of the module itself. Therefore, each module $i = 1, \dots, n$ has a rely/guarantee specification of the form: *assuming* the environment of C_i behaves as in E_i, we can *guarantee* that the module behaves as in M_i. Let C be the system obtained by composing the n modules together. In general, C has its own environment it interacts with. Let E be the assumption we make on C's environment and M the global property we want to prove of C. Therefore, C is characterized by a *global* rely/guarantee specification of the form: assuming the environment of C behaves as in E, we can guarantee that the composite module behaves as in M. The compositional inference rule lets us derive the validity of the global rely/guarantee specification of C from the validity of the local rely/guarantee specifications of the C_is.

Now, we need to introduce some formal semantics for rely/guarantee specifications; in other words, we have to define formally what is the link between the assumption formula of one module (e.g. E) and its guarantee formula (e.g. M). In order to do this, we introduce a new temporal operator, represented by the $\overset{+}{\rhd}$ symbol[6]. Let P and Q be two time-dependent TRIO formulae. We

[6] The $\overset{+}{\rhd}$ symbol appears also in [1], among other works, but with different semantics.

define $P \overset{+}{\rhd} Q$ to be a shorthand for the formula:

$$P \overset{+}{\rhd} Q \quad \triangleq \quad \begin{cases} AlwP(P) \Rightarrow AlwP_i(Q) \wedge NowOn(Q) & \text{if } \mathit{Time} \text{ is dense} \\ AlwP(P) \Rightarrow AlwP_i(Q) & \text{if } \mathit{Time} \text{ is discrete} \end{cases}$$

where two definitions are given, depending on whether the temporal model we use is dense or discrete. The informal meaning of the operator is simple: $P \overset{+}{\rhd} Q$ means that Q lasts at least as long as P does, and even a bit longer.

The semantics of the link between the assumption formula and the corresponding guarantee formula of one module is given by the operator $\overset{+}{\rhd}$: for example, the global rely/guarantee specification for the module C can be written as $E \overset{+}{\rhd} M$. Basically, this formula states that when a failure of the environment occurs (i.e. E becomes false) for the first time, the module may stop respecting its specification M only "a bit later" than the occurrence of the aforementioned failure.

Now, we can formulate the compositional inference rule, which is founded on theorem 3.1 given below.

Theorem 3.1 (rely/guarantee inference rule) *If, for $i = 1, \ldots, n$ (finite) the following conditions hold:*

(i) $Som(AlwP(E_i))$

(ii) $Alw\left(E \wedge \bigwedge_{j=1,\ldots,n} M_j \Rightarrow E_i\right)$

and $Alw\left(\bigwedge_{j=1,\ldots,n} M_j \Rightarrow M\right)$, *then*

$$Alw\left(\bigwedge_{j=1,\ldots,n} (E_j \overset{+}{\rhd} M_j)\right) \Rightarrow Alw(E \overset{+}{\rhd} M)$$

The inference rule works as follows: if we are able to prove the hypotheses of theorem 3.1, and if, for all $i = 1, \ldots, n$, the module C_i respects the local rely/guarantee specification $E_i \overset{+}{\rhd} M_i$, then we can soundly infer that the composite system C satisfies the global rely/guarantee specification $E \overset{+}{\rhd} M$.

4 The PVS Encoding of Modular TRIO

For a verification framework to be usable, supporting tools are of crucial importance. This section presents the PVS-based tool built around the inference rule defined in Section 3. In particular, this section describes an encoding in the logic of the PVS theorem prover [7] of the *modular* features of TRIO [7]. The encoding is composed of two parts: a mapping of the modular features of TRIO onto the PVS language (Section 4.1), and a set of strategies that automate the conduction of PVS proofs of TRIO specifications (Section 4.2).

4.1 The Mapping of Modular TRIO onto PVS

Each TRIO class is mapped onto a PVS *theory*, the basic PVS encapsulation mechanism. Genericity is also translated naturally from TRIO to PVS: in fact, both TRIO classes and PVS theories can be generic with respect to a number of parameters. Therefore, each TRIO class parameter maps onto a PVS theory parameter; for example, TRIO constants are mapped onto PVS constants and TRIO domains are mapped onto PVS types.

Furthermore, an additional parameter must be added to each PVS translation of a TRIO class. This parameter is named `instances` and is a non-empty type (in PVS: `TYPE+`). Whenever we translate a TRIO item into PVS, we add an argument of type `instances`. For example, a TRIO time-dependent proposition `I`, which for TRIO in-the-small would translate to a PVS item `I: TD_Fmla`[8], is instead defined in PVS as `I: [instances -> TD_Fmla]`, that is a function from `instances` to `TD_Fmla`. This particular way of parameterizing theories is needed to render in PVS the TRIO semantics of module importing. In fact, PVS does not allow importing multiple instances of the same theory, while TRIO does. This problem is solved by importing multiple instances of the same PVS theory using different actuals for the `instances` parameter (one for each corresponding TRIO module).

Example 4.1 [The reservoir system in PVS] Let us consider the translation of the reservoir system in PVS. First of all, the reservoir_system class is mapped onto the following PVS theory (similarly for classes reservoir and controller):

```
reservoir_system [instances: TYPE+, fr: posreal, ...]
    : THEORY
```

Then, we declare two types that are used to identify the instances of the reservoir and controller classes (i.e. PVS theories).

```
Res_type: TYPE = {n: nat | n = 0} CONTAINING 0
Ctl_type: TYPE = {n: nat | n = 0} CONTAINING 0

IMPORTING reservoir[[instances, Res_type]],
```

[7] Our encoding is based on the results presented in [4], which deal with the in-the-small features of TRIO.

[8] `TD_Fmla` is the PVS representation of time-dependent formulae, as defined in [4].

```
          controller[[instances, Ctl_type]]
```

Then, theorem level_stays_between_bounds of class reservoir_system (see Example 2.4 in Section 2) is translated into the PVS formula shown below.

```
Res: VAR [[instances, Res_type]]

level_stays_between_bounds: THEOREM
Alw( reservoir.level(Res) >= L_l
     AND reservoir.level(Res) <= L_u )
```

Two important features of the TRIO language, namely inheritance and visibility management, cannot be translated properly into correspondent PVS constructs. In fact, while PVS does have some mechanisms to support the reuse of code and to perform a minimal information hiding, they simply are too weak and not flexible enough to represent effectively the corresponding TRIO features. Therefore, inheritance and visibility management should be entirely realized by front-end tools that would serve as interface between the user and the PVS engine. Basically, these tools would ensure an automatic translation of TRIO code into PVS code, respecting the TRIO semantics for these important object-oriented features. Such tools are in fact currently under development. For the sake of brevity, we do not illustrate with examples the limitations of the PVS constructs in translating TRIO inheritance and visibility; the interested reader can find them in [3].

4.2 PVS Strategies for TRIO Proofs

A PVS proof strategy is a script that can automate frequently occurring passages of proofs, aiding the management of PVS features and of low-level details of the mapping from TRIO. This section briefly describes PVS strategies built to help the conduction of PVS proofs of TRIO modular specifications, and particularly of rely/guarantee specifications.

The first feature we have to manage effectively during PVS proofs is the use of the instantiation parameters that separate distinct TRIO modules in PVS (see Section 4.1). Since each item of a PVS theory corresponding to a TRIO class is parametric with respect to a variable of `instances` type, whenever we manipulate formulae in PVS, either to prove them, or to use them to derive other properties, we have to replace those generic variables with Skolem variables of the same type. Doing this both for the antecedent and for the consequent formulae in a PVS sequent, the prover acknowledges we are referring to the same items, and can validate the proof. Therefore, some strategies aim at reducing the user interaction needed to handle these frequently occurring instantiations. Basically, when starting a proof we immediately store the newly introduced Skolem variables into a persistent table. Afterwards, whenever another formula is introduced in the PVS sequent, other strategies reuse the values we have stored to perform the necessary instantiations transparently. In particular, we can store multiple sets of instantiation values and then decide which set to use to perform instantiations for a given new formula. Combining these basic strategies with other strategies implementing heuristics for instantiations of temporal values, we built commands that are often able to present a formula in a conveniently usable form, without explicit user interaction.

Some other strategies are specifically tailored at simplifying proofs of systems specified with the rely/guarantee framework of Section 3. First of all, a PVS theory declares what is needed to translate the $\overset{+}{\rhd}$ operator and the rely/guarantee inference rule of Section 3 in a way that is conveniently usable during proofs. This theory must be imported in every other theory performing rely/guarantee reasoning. In order to translate the sets of formulae E_is and M_is into PVS, we declare items of type $[\{i \in \mathbb{N} : 1 \leq i \leq n\} \rightarrow \texttt{TD\_Fmla}]$, and associate each value of i with a module of those we are composing. As a result of this practice, we often have to split the proofs into n branches, one for each of the modules. Some strategies realize this splitting in a convenient way, and, by combining the usual PVS simplification heuristics with information about the indexing of the modules, can often close each branch without requiring further user interaction.

5 Case Study

The case study we consider is a controlled reservoir system, introduced in Section 2. In this section, we briefly review some methodological aspects arisen from the development of the specification of this case study (Section 5.1), including the application of the rely/guarantee framework of Section 3. Moreover, we outline the proof of the global correctness property level_stays_between_bounds (see Section 2), which was demonstrated using the previously introduced compositional inference rule (Section 5.2).

5.1 Methodological Considerations

The specification of the reservoir system has been the result of a two-step refinement process, exploiting TRIO inheritance mechanisms. The first version of the classes describes the basic behavior of the components, by means of axioms only. No assumptions on the environment are made and no high-level properties are derived, so that this basic kernel is as reusable as possible. The second version of the classes specifies with more detail the behavior of the components, also assuming certain constraints on the environment (i.e. the other components) interacting with the module. Derived properties are stated, relying on the environment assumptions, like rely/guarantee properties. We believe that this basic two-phase scheme to develop a class can be fruitfully applied to the specification of components in general. Obviously, while the first version of the specification is usually generic enough to be unique, there may be several different second versions, according to different operative scenarios. Moreover, multiple-step refinements are a natural extension of this basic scheme, with each step adding as much detail as it is needed.

Among the derived properties of the system, an important role was played by continuity and non-Zenoness [4]. In fact, thanks to the axioms describing the behavior of the level item, stated in Example 2.2, and by the characterization of a state item, we can prove that level is piecewise affine[9] as a function of time. This result allowed us to guarantee the validity of a property "one time step longer" in the future (as in the $\overset{+}{\rhd}$ operator).

The key step in the application of the compositional framework of Section 3 is the choice of the formulae to serve as assumptions and guarantees of the modules. A good choice is one that effectively distributes the burden of the verification of the global properties among the classes of the composite system, while minimizing the coupling among modules. Under this respect, the application of the framework seems simpler and more "natural" whenever there is some sort of feedback relation among items of the composed modules. In particular, the application of the $\overset{+}{\rhd}$ is simpler in these cases, because the feedback action ensures the validity of a property in the future as a reaction to another property holding in the past. Notice that most controlled systems exhibit this sort of relation, and the reservoir system in our example is no exception.

Let us finally consider the role of *TRIO assumptions* (i.e. temporally closed postulates) in a rely/guarantee specification. As discussed in Section 3, a rely/guarantee specification of a component is basically reducible to a single formula of the kind $E \overset{+}{\rhd} M$ where we name "assumption" the E formula and "guarantee" the M formula. E and M can be arbitrarily complex formulae, possibly composed of some temporally-closed sub-formulae as well. Whenever temporally-closed formulae take part into a rely/guarantee specification, we can postulate them using the TRIO keyword `assumption`. Then, when composing the class with other modules, these TRIO assumptions must be discharged (i.e. proved) from the formulae of the other components in the system. If we are able to discharge all the assumptions, then all the derived properties of the system (and in particular the applications of the rely/guar-

[9] An affine function $f(t)$ is such that $\exists k, c \in \mathbb{R} : f(t) = k \cdot t + c$.

antee inference rule) are guaranteed to be sound. Methodologically speaking, this way of proceeding can ameliorate the organization of the proofs and of the specifications.

5.2 The Application

This section outlines the application of the rely/guarantee framework of Section 3 and sketches the proofs of the most remarkable properties.

The reservoir class introduces an assumption on the behavior of the filling action: if the reservoir is full, no filling action is issued (formula no_filling_when_full, which is not shown here for the sake of brevity[10]). The controller class, instead, makes the following assumptions:

- level is a piecewise right-monotone function of time (level_ monotonicity);
- if a filling command is issued, the level of liquid increases (filling_raises_level);
- if the level is now greater than or equal to L_u, it will stay above L_l for the next Δ time units (delta_definition).

In addition, Δ satisfies the constraint $\Delta \leq (L_u - L_l)/lr$. Let us now consider the application of the rely/guarantee framework. The rely/guarantee behaviors of the reservoir and controller classes are formalized by the following theorems.

theorem 4 (reservoir_behavior) $\text{level} \leq L_u \overset{+}{\rhd} \text{level} \leq L_u$

theorem 5 (controller_behavior) $\text{level} \geq L_l \overset{+}{\rhd} \text{level} \geq L_l$

Let us sketch the proofs of these two theorems. The proof of reservoir_behavior is split into the two branches $AlwP(\text{level} \leq L_u) \Rightarrow \text{level} \leq L_u$ and $AlwP_i(\text{level} \leq L_u) \Rightarrow NowOn(\text{level} \leq L_u)$, according to the definition of the $\overset{+}{\rhd}$ operator. The first branch is split into cases according to whether $UpToNow(\text{filling})$ or $UpToNow(\neg\text{filling})$. Notice that there are no other cases to consider, because filling is a state item. If $UpToNow(\neg\text{filling})$ the case is trivial, since the level is surely not increasing. If instead $UpToNow(\text{filling})$ we rely on assumption no_filling_when_full to deduce that in the immediate present the level cannot raise above L_u all of a sudden. Similarly, the other branch of the proof is split into cases $NowOn(\text{filling})$ and $NowOn(\neg\text{filling})$. Again, the case $NowOn(\neg\text{filling})$ is simple since the level is not raising. The case $NowOn(\text{filling})$ can instead be closed by contradiction, assuming $\neg NowOn(\text{level} < L_u)$ and combining it with assumption no_filling_when_full.

The proof of controller_behavior is similarly split into the branches

[10] Many formulae in this section are mentioned, but not actually shown, for the sake of brevity. The interested reader can find the complete formalization of the case study in [3].

$AlwP(\text{level} \geq L_l) \Rightarrow \text{level} \geq L_l$ and $AlwP_i(\text{level} \geq L_l) \Rightarrow NowOn(\text{level} \geq L_l)$. As it is always done in these proofs, we split the first branch into the cases $UpToNow(\text{filling})$ and $UpToNow(\neg\text{filling})$. Now, the simpler case is $UpToNow(\text{filling})$ since the level is definitely raising. Instead, if $UpToNow(\neg\text{filling})$ we can exploit assumption delta_definition to deduce that the leaking action is not instantaneous and we have a non-empty time interval within which level stays above L_l. The other branch of the proof is split on cases $NowOn(\text{filling})$ and $NowOn(\neg\text{filling})$. If $NowOn(\text{filling})$ it is simple to deduce that the level is increasing, using assumption filling_raises_level. The case $NowOn(\neg\text{filling})$ requires instead another case discussion, whether filling holds or not at the current time. If not, we just have to combine axiom filling_def (see Section 2) and assumption delta_definition to get to the desired result. If instead filling holds at the current time, we introduce assumption level_monotonicity and discuss the three cases, whether $NowOn(\text{level} \lesseqgtr L_l)$. In particular, the case $NowOn(\text{level} < L_l)$ arises a contradiction if combined with axiom filling_def, thus assuring the validity of the rely/guarantee local property.

Now, we apply the rely/guarantee inference rule of Section 3 to verify the global correctness property stated by theorem level_stays_between_bounds (see Section 2). Let the reservoir module be module 1 and the controller module be module 2. The assumptions and guarantees of the two modules are: $E_1 \equiv M_1 \equiv \text{level} \leq L_u$, $E_2 \equiv M_2 \equiv \text{level} \geq L_l$, $M \equiv M_1 \wedge M_2 \equiv L_l \leq \text{level} \leq L_u$ and $E \equiv \text{true}$, being the reservoir system closed. Therefore, the proof of theorem level_stays_between_bounds reduces to the steps:

(i) $Som(AlwP(E_1 \wedge E_2))$

(ii) $Alw(M_1 \wedge M_2 \Rightarrow E_1 \wedge E_2) \wedge Alw(M_1 \wedge M_2 \Rightarrow M)$

(iii) $Alw(E_1 \overset{+}{\rhd} M_1) \wedge Alw(E_2 \overset{+}{\rhd} M_2)$

(ii) is trivial because of the definitions of the $E_{1,2}$ and $M_{1,2}$. (iii) corresponds to theorems reservoir_behavior and controller_behavior that we have just proved. Finally, (i) is subsumed by an initialization axiom of class reservoir, which states $Som(AlwP(\text{level} = L_u))$, and by the constraint $L_u > L_l$.

Finally, in order to soundly conclude $Alw(M) \equiv Alw(L_l \leq \text{level} \leq L_u)$ we still have to discharge all the assumption formulae of the two composed modules. More precisely, the assumption of the reservoir class is discharged by a theorem of the controller class; conversely the assumptions of the controller class are discharged by theorems of the reservoir class (see [3] for further details). Figure 2 shows the proof dependencies in the proof of the global correctness property (solid lines) and in the discharging of assumptions (dashed lines).

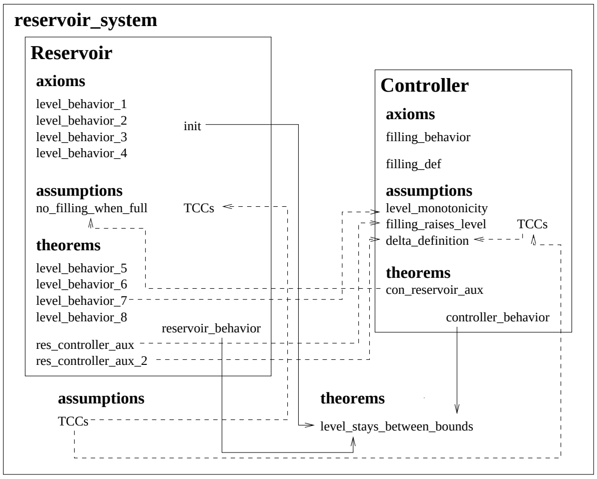

Fig. 2. Proof dependencies in the reservoir system

6 Conclusions

This paper presented a compositional, tool-supported framework for the TRIO specification language. The framework is based on an inference rule that can be used to prove that the mutual interactions between components of a complex system guarantee some property for the global application, after the components are integrated into the system. The compositional inference rule has been encoded into the logic of the PVS theorem prover and a set of supporting strategies has been developed. The PVS-based tool has been used to apply our compositional framework to an example of control system, shown in Section 5.

Future work in this line of research will follow three main directions. First, the efficacy of the compositional framework presented here will be evaluated on real-life industrial case studies. Second, automated support for the framework will be bettered and extended by improving the existing PVS strategies and creating new ones. Third, alternative, "weaker" inference rules will be investigated.

Acknowledgement

The authors would like to thank Dino Mandrioli and Angelo Morzenti for their help and support during the development of this work, and Marco Lovera for his useful suggestions and comments. We also thank the reviewers for their comments.

References

[1] Abadi, M. and L. Lamport, *Conjoining specifications*, ACM Transactions on Programming Languages and Systems **17** (1995), pp. 507–535.

[2] de Roever, W.-P., *The need for compositional proof systems: a survey*, Lecture Notes in Computer Science **1536** (1998), pp. 1–22.

[3] Furia, C. A., *Compositional proofs for real-time modular systems*, Laurea degree thesis, Politecnico di Milano (2003), online at www.elet.polimi.it/upload/rossi/Furia_LDThesis.pdf.

[4] Gargantini, A. and A. Morzenti, *Automated deductive requirement analysis of critical systems*, ACM TOSEM **10** (2001), pp. 255–307.

[5] Ghezzi, C., D. Mandrioli and A. Morzenti, *TRIO: A logic language for executable specifications of real-time systems*, JSS **12** (1990), pp. 107–123.

[6] Morzenti, A. and P. San Pietro, *Object-oriented logical specification of time-critical systems*, ACM TOSEM **3** (1994), pp. 56–98.

[7] Owre, S., J. M. Rushby and N. Shankar, *PVS: A Prototype Verification System*, in: D. Kapur, editor, *Proceedings of CADE-11*, LNCS **607** (1992), pp. 748–752.