Specification and verification of reconfiguration protocols in grid component systems.

Alessandro Basso¹
Alexander Bolotov¹
Artie Basukoski¹
Vladimir Getov¹
Ludovic Henrio²
Mariusz Urbanski³

¹ Harrow School of Computer Science
² CNRS/I3S/INRIA, Sophia Antipolis, France.
³ Institute of Psychology, Adam Mickiewicz University, Poznan, Poland.


This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Westminster’s products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

The WestminsterResearch online digital archive at the University of Westminster aims to make the research output of the University available to a wider audience. Copyright and Moral Rights remain with the authors and/or copyright owners. Users are permitted to download and/or print one copy for non-commercial private study or research. Further distribution and any use of material from within this archive for profit-making enterprises or for commercial gain is strictly forbidden.

Whilst further distribution of specific materials from within this archive is forbidden, you may freely distribute the URL of the University of Westminster Eprints (http://www.wmin.ac.uk/westminsterresearch).

In case of abuse or copyright appearing without permission e-mail wattsn@wmin.ac.uk.
Specification and Verification of Reconfiguration Protocols in Grid Component Systems

Alessandro Basso\textsuperscript{1}, Alexander Bolotov\textsuperscript{1}, Artie Basukoski\textsuperscript{1}, Vladimir Getov\textsuperscript{1}, Ludovic Henrio\textsuperscript{2} and Mariusz Urbanski\textsuperscript{3}

Abstract—
In this work we present an approach for the formal specification and verification of the reconfiguration protocols in Grid component systems. We consider Fractal, a modular and extensible component model. As a specification tool we invoke a specific temporal language, separated clausal normal form, which has been shown to be capable of expressing any ECTL\textsuperscript{+} expression, thus, we are able to express the complex fairness properties of a component system. The structure of the normal enables us to directly apply the deductive verification technique, temporal resolution defined in the framework of branching-time temporal logic.

I. INTRODUCTION

There are two approaches to building long-lived and flexible Grid systems: exhaustive and generic. The former approach provides rich systems satisfying every service request from applications but consequently its implementation suffers from very high complexity. In the latter approach, we represent only the basic set of services (minimal and essential) and thus overcome the complexity of the exhaustive approach. However, to achieve the full functionality of the system, we must make this lightweight core platform reconfigurable and expandable. One of the possible solutions here is to identify and describe the basic set of features of the component model and to consider any other functions as pluggable components [30] which can be brought on-line whenever necessary [26].

Establishing the theoretical foundations of the generic processes involved in designing and functioning of such Grid systems is highly important. A significant part of this research lies in the area of formal specification and verification of the core component model and the properties of the desired Grid systems.

Among various approaches to representing a component model we pay specific attention to the Fractal component model [17]. The advantage of the Fractal framework is that it defines the structure of the components, gives a basic classification of components, and has the mathematical foundations, e.g., the Kell calculus [7]. The Fractal specification defines the basic (non-functional) controls which should be defined especially to enable dynamic reconfiguration of components, and a number of constraints on the interplay between functional and non-functional operations. The reconfiguration is obtained by triggering appropriate actions on specific types of the components’ interfaces. These explicit dynamic properties of the Fractal component model are particularly suitable for Grid systems and environments. In this work we focus on predefined categories of reconfigurations and also on proving properties of these reconfigurations.

Given a formal specification of a distributed system there are two major approaches to formal verification of this specification: algorithmic and deductive [28]. While the algorithmic approach is fully automated, as in the case of model checking, its application, in general, is restricted to finite state systems. On the other hand, methods of the second, deductive approach, can handle arbitrary systems providing uniform proofs. To the best of our knowledge, the only technique currently used in the verification of distributed hierarchical components is model checking.

In our approach the components are modelled in a specific branching-time temporal logic, or \textnormal{SNF}_{CTL} (Separated Normal Form for Computation Tree Logic) [14], and then the temporal resolution is applied as a deductive verification tool.

\textnormal{SNF}_{CTL} initially developed for CTL has been shown to be able to express simple fairness constraints and their Boolean combinations [10], [11]. Furthermore, a clausal resolution over the set of \textnormal{SNF}_{CTL} clauses has been defined [9], [10] and recently the search strategies for this method were presented in [12].

These developments allow us to set up the following problem structure tackled in the paper:

\[
FCM \rightarrow \text{SNF}_{CTL}(FCM) \rightarrow \text{BTR}
\]

Here we suggest the translation of the Fractal component model (FCM) into the \textnormal{SNF}_{CTL}(FCM), the \textnormal{SNF}_{CTL} based formal specification of FCM, and to apply the ‘branching temporal resolution’ method (BTR), the temporal resolution technique defined over the set of \textnormal{SNF}_{CTL} clauses.

II. RECONFIGURATION SCENARIO

A. Component Model

Fractal is a modular and extensible component model. The Fractal specification defines a set of notions characterizing this model, an API (Application Program Interface), and an ADL (Architecture Description Language).

Components are characterized by their content and the membrane. The content of a component can be hidden (in which case it is simply a black box) or it can be constituted by a system of some other components (sub-
components). In the former case we would call a component \textit{primitive} while the latter case represents a \textit{composite} component. The membrane, or controller, controls the component. \textit{Controllers} address non-functional aspects of the component.

Fractal is a multi-level specifications. Depending on their conformance level, Fractal components can feature introspection and/or configuration. The \textit{control interfaces} are used in the Fractal model to allow configuration (reconfiguration), and are defined as \textit{non functional}. On the other hand, the functional interfaces of a component are associated with its functionalities. A \textit{functional} interface can provide the required functionalities and we call it the \textit{server} interface. Alternatively, a \textit{client} interface requires some other functionalities.

Component interfaces are linked together by \textit{bindings}. In this paper, we will only consider primitive bindings that are simple bindings transmitting invocations from the client interface to the connected server interface.

There are four controllers that have been already defined in Fractal (but others may be user-defined depending on the needs of the model):

- The \textit{attribute controller} is used to configure a property within a component, when there is no need to take into consideration bindings of interfaces.
- The \textit{binding controller} is used when the attribute controller is not applicable and actual binding/unbinding of interfaces is required.
- The \textit{content controller} can be used to retrieve the representation of the sub components and add or remove them accordingly; note that if a sub component is \textit{shared} by one or more other components, the scenario must be defined so that also these other components are taken into consideration.
- The \textit{life cycle controller} allows to start and stop a component, it is used for dynamic reconfiguration so that all other controls can be applied safely to the component while the component is not in execution.

These are the basic controls which should be defined especially to be able to have dynamic reconfiguration of components.

The Fractal specification defines a number of constraints on the interplay between functional and non-functional operations:

- Content and binding control operations are only possible when the component is stopped.
- When stopped, a component does not emit invocations and must accept invocations through control interfaces; whether or not an invocation to a functional interface is possible is undefined.

### B. Configuration/Reconfiguration Scenario

In general, the initial configuration of a Fractal component is given by the description of the component using Fractal ADL.

From this first state, reconfiguration is obtained by triggering appropriate actions on the the life-cycle, the binding, and the content control interfaces. A reconfiguration can be triggered by any component that has a reference to a correct non-functional interface.

In this work we focus on predefined categories of reconfigurations and on proving properties on these reconfiguration. As far as the reconfiguration is concerned we use the classical assumption that replacing a component by a similar one is safe for the system.

### III. Specification of the Scenario in Temporal Logic Framework

#### A. Formal Specification and Verification of Components

We distinguish the specification of the primitives and of the composite component. The primitives are specified as a black box, usually in a programming language of our choice. The component composition is specified using Fractal ADL, and from these specifications it is possible to extract the bindings between interfaces of subcomponents and the controllers of the component itself.

#### B. Specification Language: Normal Form for ECTL$^+$

The language of a normal form, SNF$\text{CTL}^+$ developed for a number of branching-time logics, CTL$^*$ [8], [14], ECTL$^1$ [10] and ECTL$^+$ [11], is based upon the extended set of classical logic operators $\land$, $\lor$, $\Rightarrow$, $\neg$, the set of future time temporal operators $\Box$ (always), $\Diamond$ (sometimes), $\langle$ (next time) and path quantifiers $A$ (on all future paths) and $E$ (on some future path), classically defined constants true and false, and a new operator, start (at the initial moment of time) with the intended meaning that it is true only at the initial moment of time).

**Underlying Tree Structures.** Assuming familiarity of the reader with the basic tree structure concepts we precede the presentation of the SNF$\text{CTL}^*$ language by the introduction of the notation that we will utilise.

**Definition I (Tree) A tree, $T$, is a pair $(S, R)$, where $S$ is a set of states and $R \subseteq S \times S$ is a relation between states of $S$ such that $(a) s_0 \in S$ is a unique root node (b) for every $s_i \in S$ there exists $s_j \in S$ such that $R(s_i, s_j)$; and (c) for every $s_i, s_j, s_k \in S$, if $R(s_i, s_k)$ and $R(s_j, s_k)$ then $s_i = s_j$.

By $x_i$, we abbreviate a path departing from $s_i$. A path $x_{s_i}$ is called a fullpath. Let $X$ be a family of all fullpaths of $T$. Given a path $x_{s_i}$ and a state $s_j \in x_{s_i}, \ (i < j)$ we term a finite subsequence $[s_i, s_j] = s_i, s_{i+1}, \ldots, s_j$ of $x_{s_i}$ a \textit{prefix} of a path $x_{s_i}$ and an infinite subsequence $s_j, s_{j+1}, s_{j+2}, \ldots$ of $x_{s_i}$ a \textit{suffix} of a path $x_{s_i}$, abbreviated $SuF(x_{s_i}, s_j)$.

We assume that the underlying trees are countable trees, i.e. any fullpath $\chi$ is X isomorphic to natural numbers and every state $s_i \in S$ has a countable number of successors.

Now we are ready to define the formal syntax and semantics for SNF$\text{CTL}^*$. A set of SNF$\text{CTL}^*$ clauses is interpreted in a structure $\mathcal{M} = (S, R, s_0, X, L)$, where $(S, R)$ is a countable tree with a root $s_0$. $X$ is a set of all fullpaths and $L$ is an interpretation function mapping atomic propositional symbols to truth values at each state and the following condition is satisfied: $X$ is $R$-generable [19], i.e. for every state $s_i \in S$, there exists $\chi_j \in X$ such that $s_i \in \chi_j$, and
for every sequence \(x_j = s_0, s_1, s_2, \ldots\), the following is true: 
\(x_j \in X\) if, and only if, for every \(i\), \(R(s_i, s_{i+1})\).

In our definition of an SNF\(_{\text{CTL}}\) model structure \(M\) the set of full paths \(X\) is \(R\)-generable. Therefore, following [19], it it is suffix, fusion and limit closed.

**Syntax.** First, we fix a countable set, \(Prop = x, y, z, \ldots\), of atomic propositions. The core idea of SNF\(_{\text{CTL}}\) is to represent temporal information in the following three types of constraints. Initial constraints represent information relevant to the initial moment of time, the root of the computation tree. Step constraints indicate what will happen at the successor state(s) given that some conditions are satisfied ‘now’. Finally, Sometime constraints keep track on any eventualities, again, given that some conditions are satisfied ‘now’. Additionally, to enable sound reasoning within a specific path context during the verification, we incorporate rate indices.

**Indices.** The language for indices is based on the set of terms \(\text{IND} = \{(f), (g), (h), (LC(f)), (LC(g)), (LC(h)), \ldots\}\), where \(f, g, h \ldots\) denote constants. Thus, \(E(AQ)\) means that \(A\) holds on some path labelled as \((f)\). A designated type of indices in SNF\(_{\text{CTL}}\) are indices \((LC(\text{ind}))\) which represent a limit closure of prefixes associated with \(\text{ind}\). All Formulae of SNF\(_{\text{CTL}}\) of the type \(P \Rightarrow EOQ\) or \(P \Rightarrow E\Diamond Q\), where \(Q\) is a purely classical expression, are labeled with some index.

**Definition 2 (Separated Normal Form SNF\(_{\text{CTL}}\))** A set of SNF\(_{\text{CTL}}\) clauses is a set of Formulae \(\mathbf{A} \square [ (P_1 \Rightarrow F_1) ] \) where each of the clauses \(P_1 \Rightarrow F_1\) is further restricted as below, each \(\alpha_j, \alpha_p, \alpha_c, \beta_i, \beta_m, \gamma\) or \(\gamma\) is a literal, true or false and \(\langle \text{ind} \rangle \in \text{IND}\) is some index.

\[
\begin{align*}
\text{start} & \Rightarrow k \beta_1 & \text{an initial clause} \\
\text{\(i\)} \alpha_j & \Rightarrow A \bigcirc \{ n_{i-1} \beta_{m} \} & \text{an A step clause} \\
\text{\(q\)} \alpha_p & \Rightarrow E \bigcirc \{ s_{i-1} \beta_{r} \langle \text{ind} \rangle \} & \text{an E step clause} \\
\text{\(u\)} \alpha_c & \Rightarrow A \bigtriangledown \gamma & \text{an A sometime clause} \\
\text{\(w\)} \alpha_v & \Rightarrow E \bigtriangledown \gamma \langle \text{LC(\text{ind})} \rangle & \text{an E sometime clause}
\end{align*}
\]

**C. Interpreting SNF\(_{\text{CTL}}\)**

We define a relation \(\models\) which evaluates the SNF\(_{\text{CTL}}\) clauses at a state \(s_i\) in a model \(M\). The evaluation of the classical connectives in the states is standard. Below we represent the evaluation of the temporal operators and path quantifiers.

\[
\begin{align*}
\langle M, s_i \rangle \models AB & \text{ if for each } x_{s_i}, \langle M, x_{s_i} \rangle \models B. \\
\langle M, s_i \rangle \models EB & \text{ if there exists } x_{s_i} \\
\text{such that } \langle M, x_{s_i} \rangle \models B. \\
\langle M, x_{s_i} \rangle \models \square B & \text{ if for each } x_{s_i} \text{ if } i \leq j \text{ then } \langle M, s_{j+1} \rangle \models B. \\
\langle M, x_{s_i} \rangle \models \bigdiamond B & \text{ if there exists } x_{s_i} \text{ such that } i \leq j \text{ and } \langle M, s_{j+1} \rangle \models B.
\end{align*}
\]

**Definition 3 (Satisfiability, validity)** An SNF\(_{\text{CTL}}\) clause, \(C\), is satisfiable if, and only if, there exists a model \(M\) such that \(\langle M, s_i \rangle \models C\). An SNF\(_{\text{CTL}}\) clause, \(C\), is valid if, and only if, it is satisfied in every possible model.

An initial SNF\(_{\text{CTL}}\) clause, \(\text{start} \Rightarrow F\), is understood as “\(F\) is satisfied at the initial state of some model \(M\).” Any other SNF\(_{\text{CTL}}\) clause is interpreted taking also into account that it occurs in the scope of \(\mathbf{A}\).

Thus, a clause \(\mathbf{A} \square [ (x \Rightarrow A \land p) ]\) is interpreted as “for any fullpath \(x\) and any state \(s_i \in \chi(0 \leq i)\), if \(x\) is satisfied at a state \(s_i\) then \(p\) must be satisfied at the moment, next to \(s_i\), along each path which starts from \(s_i\).

Next, a clause \(\mathbf{A} \square [ (x \Rightarrow E \land q) ]\) is interpreted as “for any fullpath \(x\) and any state \(s_i \in \chi(0 \leq i)\), if \(x\) is satisfied at a state \(s_i\) then \(q\) must be satisfied at the moment, next to \(s_i\), along a path which starts from \(s_i\) and which is associated with \(\text{ind}\).” Speaking informally, we interpret \(\mathbf{A} \square [ (x \Rightarrow E \land q) ]\) such that given a state in a model which satisfies \(x\) (the left hand side of the clause), the label, \(\text{ind}\), indicates the direction, in which the successor state which satisfies \(q\) can be reached (see similar developments in the construction of logic DCTL* [27]).

Finally, we would like to point out that our interpretation of an LC index corresponds to the concept of a linear interpretation [36].

Note that in the full ECTL* language the standard ‘until’ (\(\langle U \rangle\)) and ‘unless’ (\(\langle W \rangle\)) operators are used:

\[
\begin{align*}
\langle M, x_{s_i} \rangle \models AU B & \text{ if there exists } s_j \in x_{s_i} \text{ such that } i \leq j \text{ and } \langle M, s_{j+1} \rangle \models B \text{ and for each } s_k \in x_{s_i}, i \leq k < j \text{ then } \langle M, s_{j+1} \rangle \models A. \\
\text{and } AW B & = \mathbf{A} \forall AU B. \text{ In the SNF\(_{\text{CTL}}\) these operators are defined via the basic set of SNF\(_{\text{CTL}}\) operators [8]. For example, the following rules can be applied to remove the } \langle U \rangle \text{ operator in the scope of either of the path quantifiers [8] where } x \text{ is a new proposition:}
\end{align*}
\]

**Removal of \(\langle U \rangle\)**

\[
\begin{align*}
P & = E(p \lor q)_{\langle LC(\text{ind}) \rangle} & P & = A(p \lor q) \\
x & = E\mathbf{O}(q \lor (p \land x))_{\langle \text{ind} \rangle} & x & = E\mathbf{O}(q \lor (p \land x))
\end{align*}
\]

**D. Example Specification**

Let us consider a simple printing queue component model which consists of a client and one printing queue component as primitives. The client interfaces of the client are of type CTL\(_k\) and the server interfaces of the printing queue are of type SL\(_k\). Finally, we have a simplified version of a life-cycle controller that allows to safely add or remove a binding between a client and the printing queue.

**Formal specification of non-functional aspects.** In order to allow for reconfiguration, not only the scenario must be formally specified, but also everything else which allows dynamic reconfiguration. Although in the fractal model four controller interfaces are defined, for reasons of space, we will only specify the safe-unbinding part of a reduced Life-Cycle Controller (LCC) so that it can be used in the deductive reasoning. Note that it is always possible to create new controllers if needed, in this case an
appropriate set of formal specifications for each controller must be provided using a similar procedure. If a controller follows the standard Fractal model, a standard set of general temporal logic rules can be called and then modified to match the specification; otherwise, in the case of user-made definitions, the programmers themselves must provide the rules matching the criteria followed in the creation of the definition.

Next we will let the propositions $Bound_1, \ldots, Bound_n$ denote the bindings between components. The format that each may take is $Bound_i(CI_a, SI_r)$ (1 $\leq i \leq n$) which is a proposition that (when true) specifies that a component with Client Interface $CI_a$ is bound to the Server Interface $SI_r$. In this example we have two primitive components, one for the Printing Queue and one for the Client using the Printing Queue. We would add as many of these propositions as necessary to describe the system.

$LCC$ is a proposition which when true signifies that the Life Cycle Controller is active.

Before introducing the Life Cycle Controller Formula we would need to specify how components are started and stopped. However, for illustration in the context of this paper we will only provide a partial specification of the Life Cycle Controller and two primitive components; we only deal with the formula that captures the bindings of the two components. We will model the start of the components by attaching them to start.

Now we introduce the formula for our version of the Simplified Life-Cycle Controller:

$$\neg LCC \land \neg (Bound_1(CI_a, SI_r) \lor Bound_2(CI_a, SI_r))$$

{
\begin{center}
\begin{tabular}{ll}
(1) $\text{req} \Rightarrow A(req \land \neg print)$ & Request is kept until it is possible to execute it \\
(2) $\text{req} \Rightarrow A(\neg req \land print)$ & There will be no other request until job is printed \\
(3) $\text{req} \Rightarrow A \lor \neg req$ & The request for print will be eventually released \\
\end{tabular}
\end{center}

The complete specification of the primitive:

$$\text{start} \Rightarrow \neg \text{req} \land (1) \land (2) \land (3)$$

where $\neg \text{req}$ defines the initial state for Client primitive.

**Printing queue specification:**

\begin{center}
\begin{tabular}{ll}
(4) $A \land \neg (\text{print} \land \neg print)$ & Mutual Exclusion property: at every point in time, the printer can perform at most one printing operation: \\
(5) $A(\neg print \land \neg req)$ & There is no printing unless requested \\
(6) $\text{print} \Rightarrow A \lor \neg req$ & Printing will eventually end \\
(7) $\text{req} \Rightarrow A \lor \neg print$ & The request for printing should be granted \\
\end{tabular}
\end{center}

The complete specification of the primitive:

$$\text{start} \Rightarrow \neg \text{print} \land (4) \land (5) \land (6) \land (7)$$

Finally we specify the Life-Cycle Controller properties which affect the receiving of a printing request and the printing itself:

$$\text{start} \Rightarrow [(\neg LCC \land \neg (\text{req} \land \text{print})) \Rightarrow A \land \neg LCC \Rightarrow (\text{req} \land \text{print})]$$

When the life-cycle controller is activated, it ensures that Client Interface and Server Interface are bound, therefore allowing for requests to be sent from the Client, and prints to be carried out by the Printing Queue, for the specific binding.

We believe that the branching-time framework is appropriate for our specification targets because of the following reasons. Assume that after unbinding a client CI, it has been removed forever. Now, from this moment of time it is true to say that $A \land \neg req$ (in all possible futures from now on, there will be no more requests from the Client Interface to the Server Interface) and therefore at the previous moment of time it was true to say that $E \land A \land \neg req$ (in some future it will not be possible for the Client interface
to send a request to the Server Interface). The branching-time framework used shows how significant its use can be even in such simple examples.

To apply deductive reasoning to this model, various properties could be taken into consideration. As a relatively simple example we consider the following property. Let \( p \) stands for \( \neg \text{eq}(C_{I_0}, S_{I_0}) \land \neg \text{print}(C_{I_0}, S_{I_0}) \). Assume now that during the reconfiguration of the system the following property should be verified:

\[
\Downarrow A( \square \diamond p \land \square \neg p )
\]

In the next section we will show how this formula can be represented in terms of \( \text{SNF}_{\text{CTL}} \) and then apply to this specification the resolution technique as a verification method.

IV. The Verification Method - Clausal Temporal Resolution

A. Temporal Resolution Method for Branching-Time Logics

In order to achieve a refutation of the generated specification, we incorporate two types of resolution rules already defined in [8], [14]: step resolution (SRES) and temporal resolution (TRES). Step resolution is used between Formulae that refer to the same initial moment of time or same next moment along some or all paths. Two step resolution rules that will be used in our example are given below (where \( l \) is a literal and \( C \) and \( D \) are disjunctions of literals).

**SRES 1**

<table>
<thead>
<tr>
<th>SRES 1</th>
<th>SRES 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>start ( \Rightarrow C \land l )</td>
<td>( P \Rightarrow A \diamond (C \land l) )</td>
</tr>
<tr>
<td>start ( \Rightarrow D \land \neg l )</td>
<td>( P \Rightarrow \neg (D \land \neg l) )</td>
</tr>
<tr>
<td>start ( \Rightarrow C \land D )</td>
<td>( (P \land Q) \Rightarrow E \diamond (C \land D) )</td>
</tr>
</tbody>
</table>

When an empty constraint is generated on the right hand side of the conclusion of the resolution rule, we introduce a constant \( \text{false} \) to indicate this terminating clause.

Now we present only two temporal resolution rules that will be used in our verification example. In the formulation of these rules below \( l \) is a literal and the first premises abbreviate the \( A \) and \( E \) loops in \( l \) [13], i.e. the situation where, given that \( P \) is satisfied at some point of time, \( l \) occurs always from that point on all or some path respectively.

**TRES 2**

<table>
<thead>
<tr>
<th>TRES 2</th>
<th>TRES 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P \Rightarrow A \diamond A \square l )</td>
<td>( P \Rightarrow E \diamond E \square l )</td>
</tr>
<tr>
<td>( Q \Rightarrow E \square (\neg l) )</td>
<td>( Q \Rightarrow A \square (\neg l) )</td>
</tr>
</tbody>
</table>

B. Example Verification

To verify \( (\downarrow) \) we apply the resolution method to the set of \( \text{SNF}_{\text{CTL}} \) clauses \( \text{SNF}_{\text{CTL}}(\downarrow) \). We commence the resolution proof presenting at steps 1 - 13 the clauses of \( \text{SNF}_{\text{CTL}}(\downarrow) \) in the following order: initial clauses, step clauses and, finally, any sometime clauses.

1. \( \text{start} \Rightarrow x \)
2. \( \neg x \lor y \)
3. \( \neg x \lor z \)
4. \( \neg z \lor p \)
5. \( \neg z \lor z_1 \)
6. \( \text{true} \Rightarrow A \diamond (\neg z \lor \neg p) \)
7. \( \text{true} \Rightarrow A \diamond (\neg z \lor z_1) \)

We apply step resolution rules between 1 and 2, and 1 and 3. No more SRES rules are applicable. Formula 12 is an eventuality clause, and therefore, we are looking for a loop in \( \neg p \) (see [13] for the formulation of the loop searching procedure). The desired loop, \( E \square E \diamond (\neg p) \) (given that condition \( z_1 \) is satisfied) can be found considering clauses 10 and 11. Thus, we apply the TRES 3 rule to resolve this loop and clause 12, obtaining 16. Next we remove \( E W \) from 16 deriving a purely classical Formula 17 (\( y \) is a new variable). Simplify the latter, apply TEMP (the 'temporising' rule), see [8], obtaining, in particular, 19 and 20, and then a series of SRES rules to newly generated clauses. Now, as no more SRES rules are applicable, we find another eventuality, Formula 13, and thus we next look for a loop in \( \neg z \). This loop can be found considering Formulae 9 and 26: \( A \diamond A \square \neg z \) given that condition \( z_1 \) is satisfied. Thus, we can apply TRES 2 to resolve this loop and 13 deriving 27. Then we remove \( E W \) from the latter (on step 28, where \( w \) is a new variable, we use only one of its conclusions). Applying simplification and temporising to 28 we obtain 29. The desired terminating clause \( \text{start} \Rightarrow \text{false} \) is deduced by applying SRES 1 to steps 15 and 23.

We have found a contradiction, meaning that \( \text{SNF}_{\text{CTL}}(\downarrow) \), hence \( \downarrow \) itself is unsatisfiable.

V. Conclusions and Future Work

In this paper we have introduced a formal framework for the deductive verification of modular specification. As a specification tool we use the branching-time temporal logic. Specified properties and requirements of the system are then translated into the language of a normal form, \( \text{SNF}_{\text{CTL}} \), thus enabling the application of a powerful resolution method.
Future extensions of this work will be in the application of the Inferential Erotetic Logic (IEL) tools aiming at optimisation of the process of reconfiguration of a component model [15]. The Inferential Erotetic Logic (IEL) [31], [34] is a powerful tool in the area of analyzing and modelling such components of intelligent activity as planning, problem solving, and searching for information in massive data/knowledge bases [32]. Important developments within the framework of IEL are Erotetic Search Scenarios (ESS) [33] and Socratic Proofs (SP) [35]. ESS is based on the idea of providing conditional instructions for solving an initial problem, informing us which questions should be asked and when they should be asked. Moreover, an erotetic search scenario shows where to go if a direct answer to a query appears to be acceptable and does so with respect to any direct answer to each query. SP is a very specific technique, which reduces the complexity of standard problem-solving methods by using pure questioning only.

REFERENCES