Specifying and Verifying the SYNERGY Reconfiguration Protocol with LOTOS NT and CADP

Fabienne Boyer[1], Olivier Gruber[1], and Gwen Salaün[2]

[1] UJF-Grenoble 1, INRIA, France
{Fabienne.Boyer,Olivier.Gruber}@inria.fr
[2] Grenoble INP, INRIA, France
Gwen.Salaun@inria.fr

Abstract. Dynamic software systems that provide the ability to reconfigure themselves seem to be reaching a complexity that suggests the use of formal methods in the design process, helping system designers master that complexity, better understand their systems, find and correct bugs rapidly, and ultimately build strong confidence in the correctness of their systems. As an illustration of this trend, this paper reports on our experience with the co-design and specification of the reconfiguration protocol of a component-based platform, intended as the foundation for building robust dynamic systems. We wrote the specification in LOTOS NT, whose evolution from the E-LOTOS standard proved especially suited to this work. We extensively verified the protocol using the CADP toolbox. This formal analysis helped to detect several issues which enabled us to correct various parts of the protocol. The protocol is implemented in the SYNERGY virtual machine, the prototype of an ongoing research programme on reconfigurable and robust component-aware virtual machines.

1 Introduction

A major factor in the complexity of modern software systems is their ability to reconfigure themselves as directed by changing circumstances. This ability often relies on the component paradigm where software is understood as an assembly of components that can be reconfigured dynamically as one sees fit. While expressing a desired reconfiguration is relatively simple, actually evolving a running system, without shutting it down, is complex. This is even more complex when considering failures that may happen during the reconfiguration process.

At the heart of this reconfiguration capability lies the *reconfiguration protocol*, a protocol that is responsible for incrementally and correctly evolving a running system. This evolution happens incrementally, invoking individual reconfiguration operations on components. Therefore, a key challenge of this protocol is to compute and order the set of individual reconfiguration operations that are necessary to evolve one assembly of components into another. This is complex

because the ordering of reconfiguration operations must never violate several invariants regarding the overall structure of the evolving assembly, and must also respect a reconfiguration grammar per component. Respecting this grammar is crucial as it underlies the programming model given to component developers. In addition, failures may happen during a reconfiguration and must be handled in a way that continously respects both the invariants and the reconfiguration grammar.

Reconfigurable component-based software has been the subject of quite some work during the last decade [3, 8, 6, 7], and has made its way into most modern middleware platforms such as Eclipse, Web application servers, Web browsers, and even main-stream operating systems such as Windows or Linux. However, tolerating failures that occur during such reconfigurations remains a crucial challenge [16]. The protocol presented in this paper is the first protocol, to the best of our knowledge, to tolerate multiple failures occuring at reconfiguration time.

We designed and implemented such a protocol in the SYNERGY virtual machine, an experimental Java virtual machine that is fully component-aware and strives to guarantee robust software reconfigurations. Soon after a first version was partially running, it became obvious that the complexity of the protocol required a more formal approach, relying on specifying and verifying the protocol to help not only the design and implementation efforts but also increase the confidence of the overall robustness of the protocol.

We specified the reconfiguration protocol using LOTOS NT [4] and verified it with the CADP toolbox [9]. LOTOS NT is a simplified variant of the E-LOTOS standard [10] that combines the best features of imperative programming languages and value-passing process algebras. LOTOS NT has a user-friendly syntax, and supports the description of complex data types written using a functional specification language. This makes specifications easy to understand and write by system designers. In our case, this greatly simplified the design and analysis process. This reduced gap between the specification and the real implementation of the system drastically improved the confidence of system experts in the relevance of the verification process. Moreover, the late introduction of formal techniques and the establishment of a virtuous circle between the design, the specification, the verification, and the implementation efforts, were a success. It lowered the entry costs for specification specialists because the specification could be approached incrementally, in parallel with the design and implementation of the real system. It also helped us understand the finer points of the protocol earlier, thereby significantly reducing the implementation and testing efforts.

The rest of this paper is organized as follows. Section 2 and 3 introduce the concept of a component assembly and the reconfiguration protocol, respectively. We present the LOTOS NT specification language and the specification of the reconfiguration protocol in Section 4. Section 5 details the different checks we have done and presents some experimental results. After comparing our experience with related work in Section 6, we conclude this paper in Section 7 with the lessons we have learned.

2 Component Assembly

In the component paradigm, complex systems are designed and built as a component assembly, depicted in Figure 1. Components are independent fragments of software, assembled together by wiring imports to exports. For each component, its exports describe services that the component is willing to provide and imports describe service requirements, that is, services that the component needs to function properly. A wire from an import to an export indicates that the service requirement described by the import is to be satisfied by the provided service described by the export.

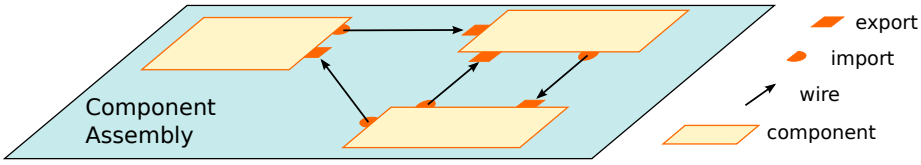

Fig. 1. A Component Assembly

To be correct, a component assembly must respect certain invariants that correlate the lifecycle of components, the different semantics of imports, and the wiring of imports to exports. There are three semantics for an import: vital, mandatory, and optional. Vital imports represent services that are needed to construct and initialize a component. Mandatory imports represent references to services that are needed by a component to be functional. Finally, optional imports express that the component may function without the corresponding services. There are four states to the component lifecycle: *registered*, *constructed*, *resolved*, and *failed*. An import is said to be *satisfied* if it is wired to an export and the component of that export is resolved. Due to space limitations, we only give below the four main invariants:

INV.1 A component is constructed if all its vital imports are satisfied.
INV.2 A component is resolved, if all its mandatory and vital imports are satisfied.
INV.3 There can be no wire from a resolved component to either a constructed, registered, or failed component.
INV.4 If a component is failed or registered, none of its exports are wired.

A component starts its life when it is registered in the assembly. It is constructed when its vital imports are satisfied. When constructed, a component has created the services it exports, but they are not yet available to use by other components. When a component is resolved, all its mandatory requirements are satisfied; it is therefore fully functional and the services it exports are available to use.

3 The Reconfiguration Protocol

The rôle of the reconfiguration protocol is to reconfigure the running system, called the *concrete assembly*. As depicted in Figure 2, the reconfiguration to apply to the concrete assembly is given to the protocol as two abstract assemblies: the *current assembly* and the *target assembly*. The *current assembly* is an abstract description of the current state of the running system. The *target assembly* is an abstract description of the desired assembly for the running system. Comparing the current and target assemblies, the protocol computes the ordered set of reconfiguration operations that must be invoked on the concrete assembly in order to reconfigure it to conform to the target assembly definition.

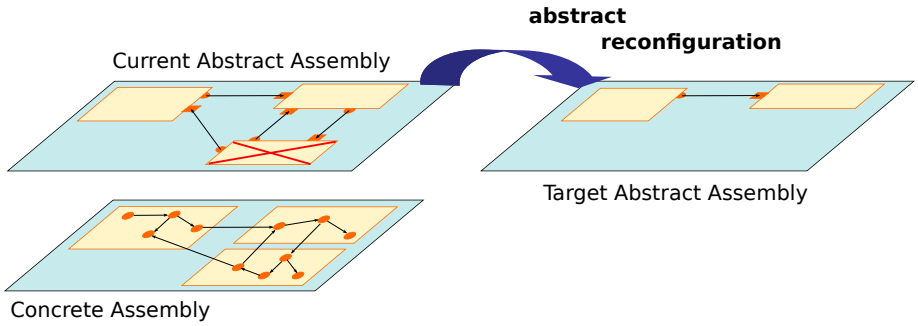

Fig. 2. Concrete and Abstract Assemblies

While computing the set of necessary operations is relatively straightforward, ordering these operations correctly is a real challenge. Correctness is defined here as (i) invariants must be respected before and after each operation, (ii) per component, the sequence of reconfiguration operations must respect the grammar corresponding to the automaton depicted in Figure 3. This correctness is crucial because it is the cornerstone of the programming model exposed to component developers. Firstly, invariants control the lifecycle of components that governs when a component is operational and when wired services may be used. Secondly, the grammar is the behavioural contract given to component developers regarding reconfigurations.

Embracing this correctness all at once is complex, so we will discuss it in three incremental steps. First, we will only consider the optional and mandatory semantics on imports, ignoring the vital semantics. Second, we will focus on the vital semantics, and third we will consider reconfiguration failures. Interestingly enough, these three steps correspond to the actual steps we followed when cooperatively designing and specifying the protocol.

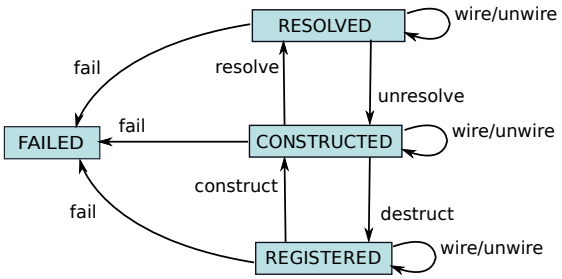

Fig. 3. Component Lifecycle

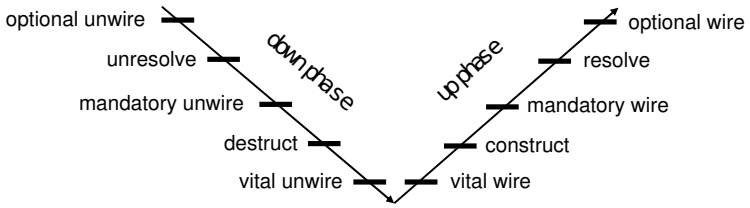

Fig. 4. Our V-shaped Protocol

Without considering the vital semantics (INV.1), the V-shape order depicted in Figure 4 is correct. During the *down phase*, it starts with *down operations* (unresolve, unwire, and destruct) applied to all components in the depicted order. During the *up phase*, it finishes with *up operations* (construct, wire, and resolve) in the depicted order. This precise order ensures that all our invariants (but INV.1) are never violated.

When considering INV.1, this V-shape ordering is no longer sufficient. INV.1 states that the vital imports of a component must be *satisfied* before that component can be constructed. To be satisfied, a vital import must be wired to a component that is already resolved. This implies that some components be resolved before some others can be constructed, however, our V-shape protocol always constructs before it resolves. To ensure that INV.1 is never violated, we must group components in different sets that we process in the correct order.

To compute these sets and order their processing, we leverage the fact that vital imports define a Direct Acyclic Graph (DAG) over an assembly of components. This DAG is useful because it splits components into layers that can be processed in distinct up and down phases of the V-shape protocol, as depicted in Figure 5. Thus, we no longer apply the down phase to all components and then the up phase to all components. We selectively apply the down phase per layer, going down in the DAG from leaf components down to the root. We then selectively apply the up phase per layer, going up in the DAG from the root

up to leaf components. At each layer, we go through the complete down phase (resp. up phase) on all components belonging to that layer.

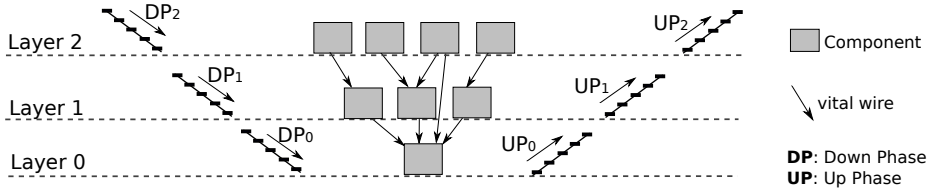

Fig. 5. Combining the V-shape Protocol with the DAG

We now consider *reconfiguration failures*: any reconfiguration operation invoked on the concrete assembly by the reconfiguration protocol may fail. Modeling such failures is important because they happen in running systems, either because of exceptional situations or bugs. It is important to insist that these failures are not failures of our protocol but the failure of individual concrete components. Our protocol resists such failures, assists the running system to recover from them, and then continues to make progress towards the target assembly.

When a component C of the concrete assembly fails to execute a reconfiguration operation, our protocol immediately suspends its V-shape processing (Figure 5) in order to recover from the occured failure. First, it marks the component C as failed. Second, it reconfigures the concrete assembly, striving to re-establish its consistency regarding INV.3 and INV.4. In other words, the impact of the failure is propagated throughout the concrete assembly, restoring all invariants. Obviously, since reconfiguration operations are invoked on the concrete assembly during this failure propagation, nested failures may occur. To cope with nested failures, the failure propagation is a fixpoint. This fixpoint terminates because the maximum number of failures is bounded by the number of components. When the fixpoint terminates, the running system has fully recovered from failures; its concrete assembly respects all our invariants.

Before our protocol can loop over on the complete V-shape protocol of Figure 5, trying to make further progress towards the target assembly, it needs to recover the consistency of both abstract assemblies. First, since the concrete assembly has been changed by the failure propagation described above, the current assembly must be changed so that it describes the concrete assembly accurately. The target abstract assembly must also be changed; the impact of component failures must be propagated throughout the target assembly, adapting it to the new reality that some components have failed. Note that failed components are not automatically repaired by this reconfiguration protocol; component repairs are managed by higher-level protocols in SYNERGY. Comparing these two modified assemblies, the protocol loops, computing a new ordered set of recon-

figuration operations and resumes the reconfiguration of the concrete assembly, evolving it further towards the new desired assembly.

4 Specification in LOTOS NT

We specified the protocol in LOTOS NT [4], one of the input languages of the CADP verification toolbox [9]. We chose LOTOS NT as our specification language because (i) it provides expressive enough operators, in particular rich datatype descriptions, for modelling the reconfiguration protocol, (ii) its user-friendly notation simplifies the specification writing, and (iii) it is equipped with state-of-the-art verification tools in order to check that the protocol works correctly.

LOTOS NT in a Nutshell. LOTOS NT [4] is a simplified variant of the E-LOTOS standard [10] that attempts to combine the best features of imperative programming languages and value-passing process algebras. LOTOS NT has a user-friendly syntax and a formal operational semantics defined in terms of labeled transition systems (LTSs). LOTOS NT is supported by the LNT.OPEN tool of CADP, which enables the on-the-fly exploration of the LTSs corresponding to LOTOS NT specifications. We give in Figure 6 the behavioural fragment of LOTOS NT we use in this paper.

$$
\begin{array}{l}
B ::= G(!E, ?x)\ \textbf{where}\ E' \mid B_1; B_2 \mid \textbf{if}\ E\ \textbf{then}\ B\ \textbf{end if} \\
\quad \mid\ \textbf{var}\ x{:}T\ \textbf{in}\ x := E; B\ \textbf{end var} \mid \textbf{while}\ E\ \textbf{loop}\ B\ \textbf{end loop} \\
\quad \mid\ \textbf{select}\ [\textbf{var}\ x_1{:}T_1, ..., x_n{:}T_n\ \textbf{in}]\ B_1\ \texttt{[]}\ ...\ \texttt{[]}\ B_n\ \textbf{end select} \\
\quad \mid\ \textbf{par}\ G\ \textbf{in}\ B_1 || ... || B_n\ \textbf{end par} \mid P[g_1, ..., g_m](E_1, ..., E_n)
\end{array}
$$

Fig. 6. Syntax of the LOTOS NT Fragment

LOTOS NT terms (denoted by B) are built from actions, sequential composition (";"), conditional ("**if**"), assignments (":="), looping behaviour ("**while**"), choice ("**select**"), and parallel composition ("**par**"). Communication is carried out by rendezvous on gates G with bidirectional transmission of multiple values (for simplicity, in Fig. 6 we consider actions with only two values being sent in both directions). Synchronizations may also contain optional guards ("**where**") expressing Boolean conditions on received values. The parallel composition operator allows multiway rendezvous on the same gate. Processes are parameterized by gates and input/output data variables.

LOTOS NT specifications can be analysed using CADP [9], a verification toolbox that has been in continuous development since the late 80s. CADP is dedicated to the design, analysis, and verification of asynchronous systems consisting of concurrent processes interacting via message passing. The toolbox contains 42 tools that can be used to make different analyses such as simulation, model-checking, equivalence-checking, compositional verification, test case generation,

or performance evaluation. CADP is widely used (760 licenses granted in 2009) and was successfully applied to real-world and industrial cases studies in many different fields such as telecommunication protocols, hardware design, embedded systems, or avionics.

The Reconfiguration Protocol in LOTOS NT. The specification in LOTOS NT consists of three parts: data types (300 lines), functions (2500 lines), and processes (900 lines). The protocol is quite small in number of lines of specification. However, it is highly complex (*e.g.*, several nested loops, see Sections 2 and 3), and its formal analysis induced numerous revisions and improvements of the protocol.

Data types describe mainly the assembly (components, imports/exports, wires, etc). Functions define first all the reconfigurations we need in the reconfiguration protocol to make the current assembly evolve towards the target assembly *e.g.*, adding/removing a wire, changing a component state, adding/removing a port, etc). Some functions also apply the failure propagation on both assemblies, and others check structural invariants that assemblies must preserve throughout the whole protocol (these functions are used for verification purposes – see Section 5). Let us show an example: the type defining the set of wires and the function `disconnect_wires` traversing these wires (`wires`) and disconnecting those connected to a given component (`cid`). We can see with this example that LOTOS NT uses the basis ingredients of the functional programming style, namely pattern matching (`case`) and recursion.

```
type TWires is set of TWire end type
type TWire is
  twire (id:TID, cexport:TID, cimport:TID, idimport:TID, idexport:TID)
end type
function disconnect_wires (cid: TID, wires: TWires): TWires is
   case wires in
   var w:TWire, l: TWires in
        nil -> return nil
     | cons(w,l) -> if (w.cimport==cid) or (w.cexport==cid) then
                      return disconnect_wires(cid,l)
                    else
                      return cons(w,disconnect_wires(cid,l))
                    end if
   end case
end function
```

Processes are used to specify the behaviour of each step in the V-shape, the failure occurrence, and the main behaviour (down and up phases applied *wrt.* the layered structure plus failure handling). Each step is specified as a LOTOS NT process which handles a specific task (*e.g.*, removing some optional wires from the current assembly, first step of the V-shape). To fulfill its task, the process calls functions to access and modify the current assembly. For verification purposes, the process body also contains some *actions* to tag some specific moments of the protocol execution such as the reconfiguration operations, a failure arrival, or the beginning of the V-shape. We show below the process `pdestruct` which

takes as input two assemblies, a list of components which need to be destructed, a Boolean indicating whether a failure occured during this step, the identifier of the component that failed, and the layer being processed (list of component identifiers). These two last parameters are output parameters. The process destructs each component of the list. For each component, the function `destruct` is called, and is in charge of updating the component state in the current assembly (`current`). We can see that for each reconfiguration, here destruction of a component, a possible failure is generated as well. One can observe some examples of actions (`DESTRUCT` and `FAILURE`) which will appear in the corresponding LTS and that will be used for the forthcoming verification of the protocol.

```
process pdestruct [DESTRUCT:any, FAILURE:any]
    ( inout current:TAssembly, target:TAssembly, lcompo:STID,
      out fail:Bool, out cfailed:TID, cl:STID ) is
  var h: TID, modif: Bool, currenttmp: TAssembly in
    while not(is_empty_stid(lcompo)) and not(fail) loop
      h:=head_stid(lcompo); lcompo:=tail_stid(lcompo); modif:=false;
      if is_in_set(h,cl) then
        eval currenttmp:=destruct(h,current,target,!?modif);
        if modif then
          select
            current:=currenttmp; DESTRUCT (!h)
            []
            FAILURE (!fdestruct of TFail,!h of TID);
            fail:=true; cfailed:=h
  end select end if end if end loop end var
end process
```

Another process is used to invoke the whole protocol (`p10`). For each step the corresponding process is called to apply the different required reconfigurations. The down and up phases are preceded by the computation of the DAG (see Section 3) which guides the order of application of the different reconfigurations. When a failure occurs, the protocol executes a LOTOS NT function which propagates the effects of this failure on both assemblies, and restarts the V-shape. The main process consists of the parallel composition between the process `pfailure` and the process `p10` implementing the protocol. Processes in `p10` (*e.g.*, `pdestruct`) may fail, and the process `pfailure` controls these failures through synchronizations on action `FAILURE`. We can see in the process alphabet various actions used to tag some specific moments of the protocol (*e.g.*, `START`, `PROPAGATE`, `FINISH`) or to retrieve some information from the assemblies being reconfigured (*e.g.*, `CHECKINVARIANTS`, `VERIFWIRE`). These actions are used to analyse the protocol, see Section 5.

```
process MAIN [UNRESOLVE:any, UNWIRE:any, REMOVEIMPORT:any,
  REMOVEEXPORT:any, FAILURE:any, START:any, PROPAGATE:any,
  FINISH:any, CHECKINVARIANTS:any, VERIFWIRE:any, ...] is
  var source, target: TAssembly in
    source:=archi_source(); target:=archi_target();
    par FAILURE in
       p10[UNRESOLVE,UNWIRE,...](source,target) || pfailure[FAILURE]
```

```
      end par
    end var
end process
```

From this specification and two assemblies (current and target), CADP exploration tools generate an LTS describing all the possible executions of the protocol. In this LTS, transitions are labelled with the actions introduced previously. Suppose a simple assembly with two components `C1` and `C2` where `C1` is resolved and `C2` is registered (current assembly). We want to add a wire between both components (we assume that available ports already exist) and resolve `C2` (target assembly). Figure 7 shows a simplified version of the LTS the protocol specification produces. We can see that `START !1` corresponds to the beginning of the protocol application and `1` indicates that this is the first time we enter the V-shape. `FINISH` is used to tag the termination of the reconfiguration protocol. In between, the assembly is reconfigured: `WIRE !W !C1 !C2` indicates that a wire identified by `W` is added between components `C1` and `C2`, `RESOLVE !C2` indicates that `C2` is resolved. Components can also fail, *e.g.*, `FAIL! RESOLVE !C2` meaning that component `C2` has failed during the resolution phase. Every failure is followed by a propagation on both assemblies (`PROPAGATE`), and in this case both assemblies become the same since the protocol finishes (`FINISH`) right after this step.

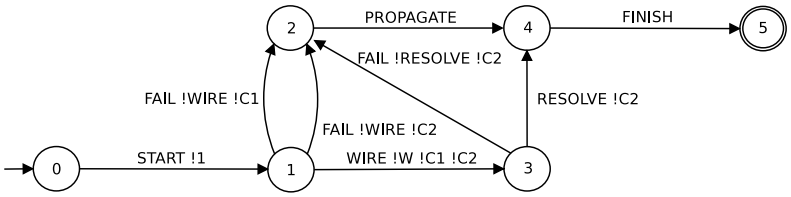

Fig. 7. LTS Resulting from the Protocol Application on a Simple System

Verification techniques presented in the next section take as input such LTSs. Depending on the input assembly, the resulting LTS may completely differ and sometimes consists of hundreds of thousands of states and transitions. For these reasons, we need some automated techniques to check that the protocol works as expected.

5 Verification using CADP

We verified the following three facets of the protocol: structural invariants, reconfiguration grammar, and temporal properties. Firstly, invariants focus on assembly structures, and we checked that all invariants are preserved throughout the whole protocol application, *e.g.,* if a component is constructed, all its vital imports are satisfied (INV.1 in Section 2). These invariants are checked using

functions which traverse the data terms storing the assemblies being reconfigured, and return Boolean values. The resulting Boolean is returned as parameter of a specific action `CHECKINVARIANTS`, and we use a simple liveness property to check that all these actions appearing in the state space never come with a *false* value. Temporal properties are verified by formalising them into μ-calculus which is the temporal logic used in CADP. We then used the Evaluator model-checker [18] that automatically says whether those properties are verified or not throughout the execution of the protocol.

Secondly, reconfiguration grammars ensure that components respect the correct ordering of actions (see Section 3) throughout the protocol. We verify for each component involved in a system under reconfiguration that its grammar is never violated. This is checked using first hiding and reduction techniques on the whole state space to keep only operations corresponding to that component. Then, we check that the resulting LTS is branching equivalent to the grammar using the Bisimulator equivalence checker [2].

These checks are important but they do not detect subtle errors that can occur in the specification such as forbidden sequences of actions. Temporal properties complement these two kinds of check by analysing the application order of operations during the protocol execution. We identified 14 temporal properties that the protocol must satisfy. Examples of such temporal properties are the following: *"if a component is constructed it is illegal to unwire vital imports"*, or *"there is no sequence where the V-shape is started twice without a failure in-between"*. Temporal properties are specified in μ-calculus and verified with Evaluator. As an illustration, the second property mentioned above in natural language is written as follows in μ-calculus:

```
[  true* . "START !*" . (not "FAILURE !* !*")* . "START !*"  ]  false
```

Experiments. Experiments were conducted on more than 200 hand-crafted examples, ranging from simple assemblies to the most pathological ones. Table 1 summarizes some of the numbers obtained on illustrative examples of our dataset. The current and target assemblies used as input to the protocol are characterized using the number of components, the maximum number of wires, and the number of reconfigurations necessary to evolve the current assembly into the target assembly. For each example, the corresponding LTS is generated using CADP by enumerating all the possible executions of the system. Verification is a time-consuming process because checking each invariant and property presented above requires traversal of the whole LTS. To reduce this verification time, we first minimize the raw LTS (using CADP reduction techniques respecting strong equivalence) to obtain an equivalent LTS where all duplicated states and paths have been removed. Hence, all verifications are performed on the reduced LTS only.

The last column gives the time to execute the whole process (LTS generation and reduction as well as checking invariants, equivalences, and properties). Experiments have been carried out on a Pentium 4 (2.2GHz, 1GB RAM) running Linux. These times grow exponentially as the number of reconfigurations

	Size			LTS (states/transitions)		Time
	components	wires	reconfigurations	raw	reduced	m:s
0010	4	5	8	115/134	44/58	1:12
0018	6	9	6	94/107	52/65	1:27
0066	9	15	13	335/401	110/157	1:54
0086	11	19	27	10,353/12,598	915/1,304	2:24
0137	16	17	11	41,386/46,758	553/671	3:37
0204	17	26	48	473,935/586,330	6,696/9,257	44:15
0207	17	28	52	875,762/1,081,136	9,964/13,873	198:22

Table 1. Experimental Results

increases. Thus, by adding only a few more reconfigurations (examples 0204 and 0207 in Table 1), the LTS is almost twice as large, and the time required for generation and verification purposes is multiplied by almost five. Fortunately, such state explosion is not a real problem in our case. Indeed, growing the reconfiguration size is much less important than covering pathological reconfiguration cases.

All the LTSs presented in this table have been obtained assuming that any reconfiguration operation on any component may fail. Furthermore, we do not consider only one failure, but all possible sequences of failures. This explains why, although our test-case assemblies are quite small, the corresponding LTSs contain up to hundreds of thousands of states and transitions. The size of these LTSs depends on the number of reconfiguration operations that need to be invoked: the more operations, the larger the resulting LTS. This also means that each failure is propagated throughout both the current and target assemblies, generating two new assemblies on which the protocol is applied again. In other words, each failure simulation generates a new test case for the protocol. Starting with 200 examples that were manually crafted, the protocol has been applied and verified over more than 2000 pairs of assemblies[3].

6 Related Work

In this section, we focus on approaches proposing formal techniques for describing and analysing dynamically reconfigurable systems. The approach proposed in our paper shares common principles with others related works that address the safety of dynamic reconfigurations through formal approaches. In particular, our V-shape ordering provides a notion of incremental consistency that is linked to the concept of a *transitional invariant* proposed in [15]. Transitional invariants are used to verify the correctness of programs during and after reconfigurations. However, in [15], such invariants are only verified on abstract specifications of programs and reconfigurations.

[3] This number has been computed experimentally by keeping track of all new assemblies generated while applying the protocol.

Our approach is also close to [20], which generates adaptive programs from formal models. Nevertheless, while our approach considers structural invariants that are application-independent, the solutions proposed in [20] focus on high-level behavioural constraints that are application-specific. Such constraints shall be individually modeled (for example using Petri nets) as well as the different reconfigurations that can be applied on the system. For each specific application, the designer can also define some properties using LTL formulas and check them on the aforementioned models using model-checking techniques.

Another set of works [12, 17, 1, 19] aims at proposing various formal models (Darwin, Wright, etc.) to specify component-based systems whose architectures can evolve (addition/removal of components/wires) at run-time. Our approach differs in at least two points: (i) we started and focused on a real implementation in Java and did not follow the classic V-shaped software lifecycle[4], and (ii) our goal in this work was mostly to verify and debug the reconfiguration protocol at hand, and not only to formalise it.

Graph grammars, in particular Reo, have been used in [14] for modeling dynamic reconfigurations of systems evolving in changing environments, and verifying properties (safety, consistency) on them. In [13], the authors also advocate the use of analysis tools to check that these changes do not affect the integrity or consistency of the system. More precisely, they show how dynamic software architectures can be specified using FSP, and some reachability and safety properties checked using LTSA. Our approach follows the same line of work, but the reconfiguration protocol is much more complex (*e.g.*, import semantics, failure tolerance, or component configuration) and therefore deserved more expressive specification languages and more powerful verification tools.

Another related work is [5], where the authors verify some temporal properties using model-checking techniques on a dynamic reconfiguration protocol used in agent-based applications. There is also a reference implementation in Java. However, analysis techniques were applied *a posteriori* on a protocol which was already working as expected, whereas we use formal verification *a priori* during the protocol design and development.

In [11], the authors present the formal verification of an operating system microkernel. They proved the functional correctness of the microkernel using the Isabelle theorem prover. The formal specification was generated automatically from an Haskell prototype, and the final implementation was manually encoded in C. This formal process helped to detect and correct many bugs in the system algorithms. Here, we focused on an alternative approach which requires much less effort in the verification process (automated versus semi-automated verification). Nevertheless, although model-checking techniques are very suitable to detect bugs in any kind of application, they do not ensure correctness of the system as it may be achieved using theorem proving techniques.

[4] This software lifecycle is completely different from the V-shaped protocol we propose in this paper.

7 Concluding Remarks

We have presented in this paper a robust reconfiguration protocol which is part of the SYNERGY virtual machine. This protocol applies a number of architectural changes to a current assembly to reach a target assembly. This protocol preserves over its application some structural invariants and is resistant to failures that may occur during the reconfiguration process. Its specification and verification helped to detect several issues which enabled us to revise several parts of the protocol, for instance: introduction of two additional (un)wire phases (a single wire/unwire was originally present in the V-shaped protocol), several corrections of the failure propagation algorithm, and several corrections in the reconfiguration grammar and structural invariants.

We think that this experience was successful due to the late introduction of specification and verification techniques in the design process (a Java implementation was already available, but was still under development). Therefore, we had several iterations between designing, specifying, and verifying the protocol on the one hand, and completing its implementation on the other hand. Through these iterations, the specification and verification refined our understanding of the finer points of the procotol, ultimately fixing bugs in the most pathological cases that would have been impossible to identify manually. In addition, this work shows that formal techniques and tools are not only of interest for criticial systems but are also necessary for the design and development of complex system protocols existing in dynamically reconfigurable systems.

Finally, we would like to emphasize that this was one of the first real-world applications of the LOTOS NT specification language. LOTOS NT, thanks to its user-friendly and programming-like notation, makes specification languages much more accessible to software engineers, and is expected to become mainstream for specifying concurrent and distributed systems.

Acknowledgements. The author would like to thank Frédéric Lang and the anonymous reviewers for their comments on a former version of this paper.

References

1. R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proc. of FASE'98*, volume 1382 of *LNCS*, pages 21–37. Springer, 1998.
2. D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*, pages 581–585. Springer, 2005.
3. É. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software – Practice and Experience*, 36(11-12):1257–1284, 2006.
4. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.1). INRIA/VASY, 109 pages, 2010.

5. M. A. Cornejo, H. Garavel, R. Mateescu, and N. De Palma. Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. In *Proc. of DAIS'01*, volume 198 of *IFIP Conference Proceedings*, pages 229–244. Kluwer, 2001.
6. G. Coulson, G. Blair, M. Clarke, and N. Parlavantzas. The Design of a Configurable and Reconfigurable Middleware Platform. *Distributed Computing*, 15(2):109–126, 2002.
7. G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A Generic Component Model for Building Systems Software. *ACM Trans. Comput. Syst.*, 26(1):1–42, 2008.
8. P.-C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *Proc. of SC'06*, volume 4089 of *LNCS*, pages 82–97. Springer, 2006.
9. H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 158–162. Springer, 2007.
10. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, 2001.
11. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. of SOSP'09*, pages 207–220. ACM, 2009.
12. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE TSE*, 16(11):1293–1306, 1990.
13. J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.
14. C. Krause, Z. Maraikar, A. Lazovik, and F. Arbab. Modeling Dynamic Reconfigurations in Reo using High-level Replacement Systems. *Science of Computer Programming*, 76(1):23–36, 2011.
15. S. S. Kulkarni and K. N. Biyani. Correctness of Component-Based Adaptation. In *Proc. of CBSE'04*, volume 3054 of *LNCS*, pages 48–58. Springer, 2004.
16. M. Léger, T. Ledoux, and T. Coupaye. Reliable Dynamic Reconfigurations in a Reflective Component Model. In *Proc. of CBSE'10*, volume 6092 of *LNCS*, pages 74–92. Springer, 2010.
17. J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proc. of SIGSOFT FSE'96*, pages 3–14. ACM, 1996.
18. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, 2003.
19. M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC / SIGSOFT FSE'01*, pages 21–32. ACM, 2001.
20. J. Zhang and B. H. C. Cheng. Model-based Development of Dynamically Adaptive Software. In *Proc. of ICSE'06*, pages 371–380. ACM, 2006.