Graph-based Redundancy Removal Approach for Multiple Cross-Layer Interactions

Ismet Aktas, Martin Henze, Muhammad Hamad Alizai, Kevin Möllering, and Klaus Wehrle
Communication and Distributed Systems, RWTH Aachen University, Germany
Email: {lastname}@comsys.rwth-aachen.de

***Abstract*—Research has shown that the availability of cross-layer information from different protocol layers enable adaptivity advantages of applications and protocols which significantly enhance the system performance. However, the development of such cross-layer interactions typically residing in the OS is very difficult mainly due to limited interfaces. The development gets even more complex for multiple running cross-layer interactions which may be added by independent developers without coordination causing (i) redundancy in cross-layer interactions leading to a waste of memory and CPU time and (ii) conflicting cross-layer interactions. In this paper, we focus on the former problem and propose a graph-based redundancy removal algorithm that automatically detects and resolves such redundancies without any feedback from the developer. We demonstrate the applicability of our approach for the cross-layer architecture CRAWLER that utilizes module compositions to realize cross-layer interactions. Our evaluation shows that our approach effectively resolves redundancies at runtime.**

I. Introduction

A promising research concept to deal with volatile network conditions such as interference, packet loss, and mobility is the cross-layer design paradigm [1], [14], i.e., the exchange of information across protocol layers and system components. Utilizing cross-layer information from other protocols and system components such as sensors can improve the performance and responsiveness of applications and protocols. However, for a developer, the process of designing and realizing even a single specific cross-layer interaction is a cumbersome task. This is mainly because the protocol stack and drivers controlling system components are integrated into the operating system which provides only a few interfaces. The process gets significantly worse for multiple cross-layer interactions. One major problem is the fact that multiple cross-layer interactions added by different developers running in parallel may lead to: (1) redundancy, i.e., multiple independently added cross-layer interactions could have common processing components wasting unnecessary CPU time and memory, and (2) conflicts, i.e., unintended interdependencies between cross-layer interactions leading to peculiar system behavior [9].

For example, assume a cross-layer interaction that provides fine-grained localization information to an application by employing both Wi-Fi and GPS related information. In contrast, another cross-layer interaction provides a coarse-grained localization information by turning off the Wi-Fi related hardware to save energy. While both interactions are designed to improve the system behavior, in this example, they may have redundant processing (e.g., parts of the localization or WiFi control might be similar) or contradicting effects (e.g., accurateness vs. energy). This is primarily caused due to the lack of development support for developers to analyze and experiment with multiple cross-layer interactions.

In this paper we focus on the problem of redundant processing of multiple cross-layer interactions and provide a general graph-based approach that automatically detects redundant parts of cross-layer interactions and removes them from the system. Our approach uses the cross-layer architecture CRAWLER [1] which is (re)configurable at runtime, i.e., allows to add, remove and change cross-layer interactions at runtime, and facilitates third-party application developers to independently insert their own set of cross-layer interactions. As a result, it is notably vulnerable to redundantly running cross-layer interactions, which can be detrimental for the system performance. Thus, cross-layer architectures necessitate the need for mechanisms, such as the one presented in this paper, that can detect and resolve such redundancies. Since CRAWLER employs module based software development (i.e., modules are composed together) to realize cross-layer interactions, redundant parts of cross-layer interactions can be found by exploring equal module compositions. To achieve this, our approach iteratively compares each pair of modules in a composition. To determine whether or not two modules are equal, it analyzes each module and its connections. Afterwards, it rewires the connections of equal modules and removes the redundant module. As this approach is based on a generic graph-based algorithm, it is not peculiar to a specific development platform and can be utilized across a wide range of modular software development systems or networking scenarios.

Overall, this paper makes the following key contributions:

- As a first step, we provide a graph-based formal description of redundant module compositions. This description is generic and relevant for a wide range of modular development platforms (Section II).
- We then propose an algorithm to automatically find and merge redundant module compositions (Section III).
- We present an extension of our approach to handle the more challenging case where cross-layer interactions change at runtime (Section IV).
- Finally, we demonstrate the applicability of our approach for the cross-layer architecture CRAWLER (Section V), and validate that it efficiently detects and resolves redundancies (Section VI).

In the remainder, we discuss prominent related work in Section VII before concluding the paper in Section VIII.

II. Generic Design

Solving the problem of redundancy in cross-layer interactions requires to analyze the semantic of the program to discover redundant functionality. Without the availability of semantic knowledge that has been provided with huge effort by developers, this is an undecidable problem according to Rice's theorem [7]. Fortunately, there exist techniques to tackle such problems and make them practically useful. By composing modules[1] to realize cross-layer interactions the cross-layer architecture CRAWLER provides a good basis to relax and tackle the problem. Before describing a concrete solution, we generalized the problem of redundancy of cross-layer interactions as redundancy in module compositions which makes our solution also applicable to other fields.

However, for our solution we opt for an automatic solution without interaction from the developer and the need for delving into program semantics because this (i) would require complex formal description of each module and its connections, and (ii) overburdens the developers with a significant effort required to support and create such descriptions. When we talk about developers, we distinguish between module developers who create the reusable modules once and module composition developers who utilize these modules to implement a certain algorithm. With our approach the former need to put effort once, while the latter are unburdened.

A. Constraints

The basic idea of our approach is to merge redundant module compositions together. But before discussing the technical details of our solution, we first discuss our primary constraints and assumptions that form the basis of our approach. For example, our constraint with regard to merging of redundant modules is that the modified system has to offer the same behavior as the unmodified system. Therefore, the basic requirement for handling modules is the ability to determine if these modules are *equal*.

Consider that we have two *equal* modules within a system, that for each input generate identical output. Any optimization that merges these modules has to fulfill two major constraints:

(1) *Output correctness* – If the merging process results in a modified composition of modules in a system, the output of the new system must be the same as the output of the unmodified system.

(2) *Modification transparency* – If a module composition developer modifies a composition of modules, the system must not require her to be aware of the merged compositions. In other words, a module composition developer should be able to parameterize modules and their compositions without having knowledge about the underlying optimized compositions.

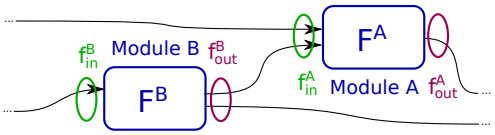


Fig. 1. Example representation of two modules A and B with their functionals and input and output functions.

B. When are Module Compositions Equal?

With respect to our constraints, to decide if a system (behavior) remains the same after merging modules, the modified and unmodified system must have the same behavior, i.e., produce the same output on a given input. In any system, the output of the system is a subset of the output of the contained modules. Thus, providing equality between the outputs of all modules in the original and the modified system is sufficient to prove the overall equality of the outputs of both systems. To achieve this, two properties need to hold for every module: (i) the behavior of the module must remain the same and (ii) the inputs of the module have to be the same.

Before describing how to verify if both of these properties hold, we give a formal description of the problem which helps to describe the idea of the algorithm and simplifies the adaptation to other networking problems. The description is based on the interpretation of a program module as a functional. As shown in Figure 1, a module M has an input function f_{in}^{M} and an output function f_{out}^{M}. The mapping of the input function to the output function is done by the *module functional* F^{M}. For example, let us consider a `Sum`-Module M with the following input function $f_{in}^{M} = (4, 1)$. Since the module calculates the the sum of its inputs, the output function is $f_{out}^{M} = (5)$.

Thus, the functional F^{M} maps an input function f_{in}^{M} to the corresponding output function f_{out}^{M}. If one can verify that two modules A and B provide the same functional, i.e., $F^{A} = F^{B}$, they both produce the same output, given the same input.

Definition *(Equality):*
A module A is equal to a module B (and thus mergeable) if and only if the following two prerequisites hold:

(1) *input equality* – the input functions of both modules are equal, i.e., $f_{in}^{A} = f_{in}^{B}$, and

(2) *behavior equality* – A and B have the same *module functional*, i.e., $F^{A} = F^{B}$.

Based on these two conditions we obtain output equality, i.e., $f_{out}^{A} = f_{out}^{B}$. In the following, we present an algorithm that checks particularly for these two conditions.

III. Graph-based Iterative Merge Algorithm

An intuitive algorithm for merging subsets of equal module compositions can be directly extracted from the requirements described in the previous section. Thus, two modules can be classified as merge-able if they satisfy the input and behavioral equality property. To achieve this, our algorithm consists of three main steps: (i) Check if two selected modules satisfy the equality property, i.e., can be merged, (ii) if yes, merge

[1] We consider a module as function that contains all the source code and variables necessary to realize a certain self-contained functionality.

them together, and (iii) repeat the previous two steps until no merges are possible anymore. Listing 1 shows our algorithm.

We now discuss the major requirements for satisfying input equality (cf. mark (1) at Listing 1), behavior equality (cf. mark (2) at Listing 1) and the merging process (cf. mark (3) at Listing 1) in more detail. Afterwards, runtime and memory consumption estimates are provided for our algorithm.

```
input: m_graph
output: m_graph
operation iterative merge begin
  changed <- true
  while (changed) do
    changed <- false
    for all (A in m_graph[modules]) do
      for all (B in m_graph[modules]) do
        if(A!=B AND
(1)       inputs_equal(A,B,m_graph) AND
(2)       behavior_equal(A,B)) then
(3)         m_graph <- merge_modules(A, B, m_graph)
            changed <- true
```

Listing 1. Graph-based iterative merge algorithm

A. Input Equality

In a real system, the equality of inputs of two modules can be assured if the following three prerequisites hold: (i) The specific connection has to originate from the same source, i.e, from exactly the same node in the module composition graph. (ii) The position of a specific input connection within the input vector has to be the same, e,g., if we consider the position of variables a and b, then $isLess(a, b)$ is not the same as $isLess(b, a)$. (iii) In a system with more than one connection type (e.g., information flow or detectable events), the type of a specific input connection has to be equal.

Prerequisites (ii) and (iii) basically enforce that the type signatures of the modules are identical, while prerequisite (i) enforces that they are always called with the same input. Combining these three prerequisites leads us to the input equality computation algorithm, as shown in Listing 2.

```
input: A, B, m_graph
output: equal
operation inputs_equal begin
  size1 = size_of(inputs_module(m_graph[inputs],A))
  size2 = size_of(inputs_module(m_graph[inputs],B))
  equal <- size1 = size2
  if(equal) then
    for all (in1 in inputs_module(m_graph[inputs],A))
      found <- false
      for all (in2 in inputs_module(m_graph[inputs],B))
        if( source_of(in1) = source_of(in2) AND
            position_of(in1) = position_of(in2) AND
            type_of(in1) = type_of(in2)) then
          found <- true
      equal <- equal AND found
```

Listing 2. Checking if two modules A and B have the same input.

B. Behavior Equality

Behavior equality in general addresses the problem if one program behaves the same way as another program does. This non-trivial property is undecidable according to Rice's theorem [7]. Since a module is just a representation of an arbitrary program, there is no assumption that holds in every module based software system to solve this problem. As we want to provide an automatic solution without investing the manual effort of semantic descriptions, the only possibility to check for behavior equality is an exhaustive state search. This exploration for each possible state within the complete state space may lead to the state-explosion problem [4]. Thus, this only works for applications with a finite and specified runtime. Moreover, if the exhaustive state search has to be performed for any possible input, it becomes infeasible and useless in most cases. To overcome this issue, we identify a possibility to relax this problem. We assume that there are only deterministic modules in the system. Thus, one specific instruction set which is running on one specific state produces exactly the same output on every run. This allows us to base the decision for behavior equality on the following two conditions.

Definition *(Behavior Equality):*
Two modules A and B have an equal behavior if

(1) *module type equality* − the implementations for both modules are exactly the same, and

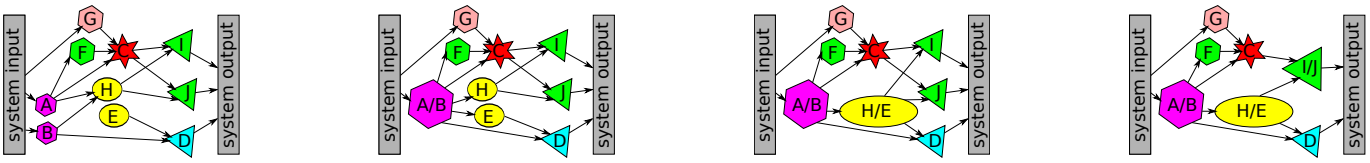
(2) *module state equality* − the current variable allocation and execution position, i.e., states of the modules, are identical.

Due to the determinism assumption, equal state and type implies the exact same behavior on any equal input. Note that the reverse direction for this implication does not hold. Since we cannot provide an equivalent property to behaviour equality, there may be cases where $F^A = F^B$ holds, which can not be found with our algorithm. However, it will not lead to false optimizations, but only oversee merging possibilities related to modules offending our assumptions. Furthermore, we assume that it is very unlikely to have two different implementations of exactly the same functionality. By introducing this relaxation, we identified a sufficient condition for behavior equality of two modules A and B, which can be practically implemented.

Although implementations of both conditions are application specific, module type equality can often easily be checked by introducing a numeric type identifier or comparing the memory. We suggest that the comparison of module states should reside inside the module since module developers know best about their modules and relevant states. Accordingly, module developers can build in a module state comparison function to verify if the module's state is equal to the state of another module. This only has to be done once and the effort to implement such a state comparing function should be reasonable. Afterwards, module composition developers (who are relieved from putting effort) can utilize modules and combine them without considering possible optimizations as our approach will automatically optimize the entire module compositions in the whole system.

C. Merging Modules

Once the algorithm finds two modules that satisfy the *behavior equality* and the *input equality* prerequisites, they can safely be merged. Merging two modules A and B is straightforward (cf. Listing 3). Module A is removed together with all its ingoing connections (as these are already provided by module B). The outgoing connections of module A are then rewired to be outputs of module B.



(a) Not merged. (b) Merged A and B. (c) Merged H and E. (d) Merged I and J.

Fig. 2. An run of our algorithm on an example module composition. The type of shapes defines the type of the modules, while the colors define its state.

```
input: A, B, m_graph
output: m_graph
operation merge_modules begin
  for all (inc in in_connections(m_graph[inputs],A))
    delete_input(inc)
  for all (outc in out_connections(m_graph[inputs],A))
    replace_source_of_connection(outc,B)
  remove_module(A)
```

Listing 3. Merging two modules A and B and removing A afterwards.

D. Runtime and Memory Consumption

The runtime of our algorithm is determined by its three nested loops (cf. Listing 1) and the three marks. The two *for* loops iterate twice over all modules in the graph, i.e., $\mathcal{O}(|V|^2)$ with V being the set of modules. Within these two *for* loops each time the three marks are checked. At mark `(1)`, the `input_equal` function, (cf. Listing 2), also iterates over two *for* loops. It first iterates over the connections in A and afterwards over all connections in B where it compares three parameters and accordingly has a runtime of $\mathcal{O}(|E|^2)$ with E being the set of connections. Mark `(2)` has a runtime of $\mathcal{O}(1)$ where the type and states are compared. The merging at mark `(3)` has a runtime of $\mathcal{O}(|E|)$ where connections are removed or rewired. If an equal module is found, all these operations are conducted again for the remaining modules. Therefore, the outer *while* loop requires a runtime of $\mathcal{O}(|V|)$, accordingly leading to a total runtime of $\mathcal{O}(|V|^3 \cdot |E|^2)$ for our algorithm.

With regard to the space complexity, our algorithm has no recursive calls that would increase the stack size of the program. Traversing lists requires the amount of space that is asymptotically equal to its length, i.e., $|V|$ for lists of modules and $|E|$ for lists of connections, i.e., in total $\mathcal{O}(|V| + |E|)$.

However, in a runtime-adaptable system, where modules and their compositions are unloaded, it is sometimes necessary to split modules again. This can happen, e.g., when the input or behavior of a merged module is changed at runtime. The module has then to be split (similar to the copy-on-write paradigm) again. How our approach handles this splitting is described in the next section.

IV. Runtime Adaptation

After discussing the challenges of the behavior and input equality constraints, and formulating an appropriate solution for it, we now turn our focus towards the *modification transparency* constraint: It requires the optimizations induced by the merging process to remain transparent to the module composition developers who intend to manually modify module compositions. Although this desired transparency constraint is inherently achieved in static systems by providing *output correctness*, runtime-adaptable systems pose further challenges: E.g., to ensure *modification transparency*, the set of commands required to achieve a desired reconfiguration of the system has to be the same before and after the merging process.

However, in a runtime-adaptable system this is difficult to guarantee because of two types of reconfigurations. First, the reconfigurations that occur due to a change in the functionality of modules. These kind of reconfigurations could occur, for example, when removing modules, modifying the state of a module, or modifying the function of a module. Second, the reconfigurations that occur due to a change in module compositions. Such reconfigurations could occur, for example, when modifying incoming data connections of a module (e.g., adding new connections or modifying connection properties).

Such configurations are usually done in an adaptation engine encapsulating the construction logic which sends commands to the adaptable software realizing the actual implementation [12]. Commands are instructions on how to modify or rather configure the adaptation software such as creating an object and connecting it with other objects. Our redundancy removal system can be placed in between these two parts to track commands. After receiving a command C, our redundancy removal system can either modify the command to fit it to the optimized version of the system obtained after the merging process, or revert specific merging optimizations to allow C to be processed normally.

A. Challenges when adding/removing Modules & Connections

The challenging nature of the commands that induce modifications on modules that have been merged is visualized in an example depicted in Figure 2. The shape of the modules represents the type while the color represents its state. For the sake of simplicity, we assume all data connections to be equal with respect to their type, ordering and any other feature. We can see that the modules A and B can be merged after validating state, type, and input equality. Similarly, the next iterations of our algorithm will also merge modules H and E and afterwards I and J.

Now let's assume a system that consists of the module compositions shown in Figure 2. *Modification transparency* requires that any developer who changes the module compositions does not need to know about the underlying merging optimizations. Thus, the developer assumes working on the first composition of modules (cf. Figure 2(a)) even though the system has been optimized by the merging process resulting in modules compositions shown in Figure 2(d). In this example, the module G can be modified by a developer without violating *output correctness* or introducing any further ambiguities. However, any modifications either to the input of module A or its functionality, would change the output of A. Clearly, this modification should not have any effect on module B.

(a) Original state. (b) Merged state. (c) Possible splitting mistake. (d) Another splitting mistake.

Fig. 3. Possible problems when not providing a reasonable history of the connections and splitting multiple modules.

In order to avoid modifying node B, there is a need for the ability to split both modules again. However, the merging process has also merged modules H and E considering that A and B are mergeable (cf. Figure 2(b) and 2(c)). Since this prerequisite will no longer be valid after the modifications introduced by the developer to module A, we have to split H and E as well. As modules I and J have been merged independently of H and E, it is not required to split I and J.

B. Splitting Affected Modules

Splitting modules requires to maintain the knowledge about both the outgoing connections and the respective modules before the merging process. We propose a simple two step procedure which has to be performed before merging modules. In the first step, we identify all those modules that were merged with a certain module A due to their equality. In the second step, we split module A and all the merged modules that were identified in the previous step by restoring the original configuration of the system, i.e., reverting back to the original connections. For this purpose, we also need a mechanism to be able to access the previously defined connections. Otherwise, the splitting process could end up in misconfigurations.

To clarify this requirement, let us consider the example in Figure 3. If there are two or more levels of connected modules that can be merged as shown in the Figures 3(a) and 3(b), the original source of data connections can vanish. In Figures 3(c) and 3(d), two different compositions are shown that could result due to the lack of information about the connections within the original system. Both wrong compositions result from splitting the merged state in Figure 3(b). Maintaining information about all the original connections is dependant upon the actual implementation of the system, and hence, beyond the scope of discussion in this paper. Nonetheless, for the sake of completeness and practical applicability, in the following section we do provide a domain specific solution to demonstrate a full fledged implementation of our mechanism for a cross-layer interaction architecture.

V. Domain Specific Use case: Cross-Layer Design

After providing the theoretical basis for our approach, we now demonstrate its applicability in the domain of cross-layer interaction using the cross-layer architecture CRAWLER [1].

In CRAWLER many cross-layer interactions can run in parallel and be added/removed/modified at runtime. To achieve this, CRAWLER utilizes modules to coordinate local information, e.g., from the network protocol stack and system components (e.g., sensors, battery, devices, etc.). By combining these modules any kind of cross-layer interaction can be achieved, e.g., switching to a more suitable TCP congestion control algorithm [1] or a audio VoIP codec [2] depending on the network conditions.

CRAWLER consists of two main components: The *logical component* (LC) resides in user space and allows cross-layer developers to express their monitoring and cross-layer interaction requirements in an abstract and declarative way. For this purpose, CRAWLER offers a rule-based language customized to cross-layer design purposes. Using this language, developers can specify cross-layer interaction at runtime and at a high level of abstraction without needing to care about implementation details. The LC further offers a uniform interface to applications for (i) providing their own set of cross-layer interactions, and (ii) exchanging information with the protocol stack and system components. However, the high level description of cross-layer interactions given by an application or preconfigured in CRAWLER are parsed and subsequently mapped to commands (as described in Section IV). Commands are instructions about how the modules are composed with each other. The commands are stored in a repository to keep track of all adaptations in the system.

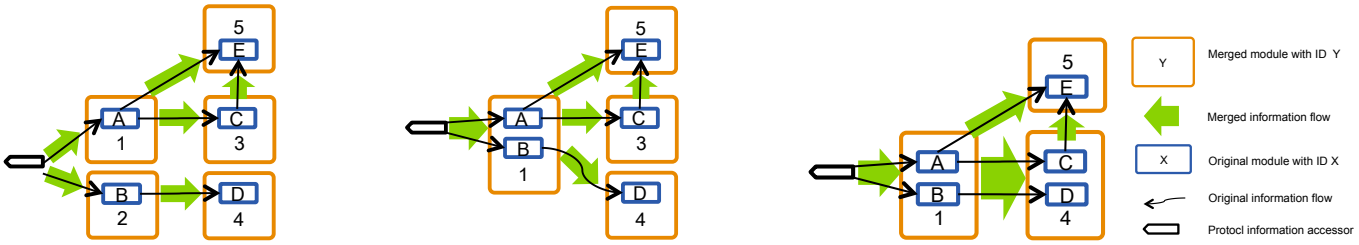
The *cross-layer processing component* (CPC) resides in the kernel space of our Linux implementation and receives the commands from the LC. Modules in CRAWLER are stateful, keeping their private variables between calls and have a uniform interface to simplify their composition. Finally, *stubs* provide read/write access to protocol information and subsystem states. In the following we show a mapping from our general approach to CRAWLER specific properties.

A. Mapping of Equality Parameters to CRAWLER's Needs

The mapping of our general equality algorithm requires to map the three functions: (i) input equality, (ii) type equality and (iii) state equality.

Regarding **input equality**, CRAWLER provides two different types of input (or connections) between modules: (a) a notify connection that is an event-based signaling and (b) a query connections that is a polling mechanisms. Redirecting connections are implemented as simple pointer redirections in C. While *notify connections* can be directly mapped onto outgoing data connections of the type `notify`, query connections have to be reversed. A query connection from one module A to another module B is realized by making A ask B for a piece of information. Thus, in reality information flows from B to A. Therefore, we interpret an outgoing query connection as an incoming data connection of type `query`. Furthermore, since the order of query connections matters, we have to take that ordering into account.

Implementing **type equality** checking in CRAWLER is straightforward, since each module carries an identifier. For example, an object of the `AND` module contains the numerical identifier `13` indicating the type. Accordingly only this numerical value has to be compared for type equality.



(a) Each original module (orange) corresponds to exactly one merged module (blue).

(b) The module A and B are realized by the module with merged ID 1.

(c) Additional iteration of our algorithm also merges original modules C and D to the merged ID 4.

Fig. 4. By reading off all original modules and connections the original composition can be extracted. By only considering the modules that reside in the CPC, one can get the *merged* composition which consists of only 3 modules. This example is restricted to query connections.

In contrast, **state equality** is a bit more complex. Since a comparison based on the internal state has to be provided by the module, each module developer has to decide how to implement the function that determines equality. CRAWLER's standard module structure predefines such a state equality interface which is called by our merge algorithm. For example, the `History`-module stores an amount of values. When the equality interface is called, the `History`-module compares the values given in its signature with the stored values. Note, the module developer knows best about its module and accordingly needs to think once about the state equality. Afterwards, the developer of cross-layer interactions can utilize them without needing to care about our merge algorithm.

So far, we know how to determine equality of modules in CRAWLER, but we still need to deal with runtime adaptation. In the following we show how we extended CRAWLER to handle runtime adaptation of module compositions.

B. Handling Runtime Adaptation

As discussed in Section IV, if module compositions change at runtime, we need to be able to keep track of the original module composition. We use CRAWLER's repository residing in the LC to keep track of such adaptations. Remember, if a developer changes a configuration, these changes are translated into commands that are delivered both to the repository and CPC. Here, the repository has two advantages: (i) It behaves similar to a revision control system: Each time the configuration changes, the commands are automatically committed as a new revision. (ii) Unnecessary context switches between user and kernel space are avoided. As a result, the repository provides a good overview of running compositions and allows the developer to roll back to a previous cross-layer interactions if necessary.

We extended the representation of modules and connections in the architecture to keep track of modules both in the original and merged compositions. For this purpose, we store two different modules IDs for two different views: (i) The *original view* consists of original modules and their original connections which in fact represents the initial and unmodified composition graph. (ii) The *merged view* is used for handling the merged version of module compositions, i.e., the optimized graph. For clerity reasons, the concept of the original and merged views is combined in one graph as shown in Figure 4. Additionally, for the sake of simplicity, we only consider one type of connections in this example representing the information flow in the graph.

Each node in Figure 4 represents one merged module which encapsulates several original modules. Similarly, a connection represents a merged connection encapsulating several original connections. The original module and connections are necessary to recreate the original module compositions. Figure 4(a) shows the module compositions in the initial unmodified state. In Figure 4(b), the modules with ID 1 and 2 are merged and their original IDs A and B are stored in the module along with one of the merged module IDs, in this case ID 1. Similarly, in Figure 4(c), modules with IDs 3 and 4 are merged along with their connections, in this case the merged module with the ID 4 is used for merging the original modules C and D.

In the following we show a real-world example of how our algorithm merges and splits module compositions automatically after a runtime adaptation occurs.

VI. Evaluation and Validation

To verify the correctness of our proposed algorithm, we performed a complete system test. We loaded the CRAWLER architecture with an empty initial configuration, i.e., no cross-layer interactions were running in the beginning. Then we started two applications that use CRAWLER's API to feed cross-layer interaction into the system. Although the naming of modules and the overall compositions differ for both applications, some parts are equal and thus we expect that our algorithm will merge them. However, later on we will modify one application's module composition at runtime and accordingly expect that our algorithm splits affected compositions and conducts the necessary modifications. In the the following we show the cross-layer interaction for the two applications.

Cross-Layer interaction of application 1: The first application is interested in knowing whether the current received signal strength indicator (RSSI) of the wireless connection is good or bad. The RSSI is measured every 250 msec. To reduce the amount of oscillation in this signal, the base for the decision is calculated by averaging over the last three values. If the result does not exceed a threshold of 80, the signal is defined as being bad. The resulting module composition is shown in Figure 5(a).

Cross-Layer interaction of application 2: The configuration used by the second application implements a simplified version of a cross-layer interaction for the TCP congestion

(a) Module composition for cross-layer interaction 1.

(b) Module composition for cross-layer interaction 2.

Fig. 5. Module composition of two installed cross-layer interactions. Red border indicates equal composition detected and merged by our algorithm.

control algorithm [1]: It monitors the RSSI of the wireless device and freezes the congestion window size (CWND) in TCP if the system operates under bad WiFi conditions. This prevents the congestion control algorithm to reduce its congestion window and thus entering the slow start phase. As shown in [5], the slow start of TCP is unsuited for short-term disturbances on the physical layer. To stop the CWND from trying to adapt to the conditions, a 0 is written to the stub accessing the corresponding TCP variable. The composition for this example is visualized in Figure 5(b).

Runtime Modification: Application 1 feeds its cross-layer interactions using CRAWLER's API into the system. Shortly after, Application 2 also feeds its cross-layer interaction into the system. Our algorithm automatically detects equal compositions. Due to the given input, state and type equality, the algorithm merges the chains starting at `app1less` and `app2less` respectively. Figure 6(a) depicts the result after the merging process.

After Application 2 has inserted its cross-layer interaction, Application 1 adapts its interaction. One possible reason could be that the resulting information about the link still jitters too often. To smoothen the output, Application 1 replaces the history module, which stores three values, by another module that stores ten values. In CRAWLER this runtime adaptation could easily be expressed with the following two lines:

```
app1newhistory:history(app1rssi,10)
replace(app1history,app1newhistory)
```

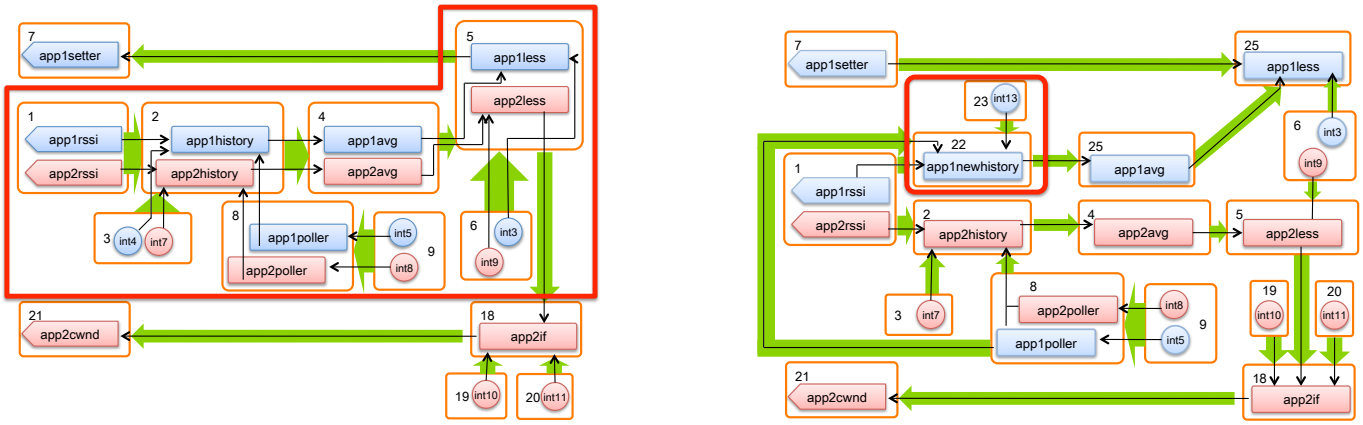
The first line creates a new history module with 10 elements and the second line instructs to exchange the old history module with the new one. When Application 1 tries to exchange `app1history` by `app1newhistory`, the connections (incoming notify connections and the outgoing query connections) of `app1avg` are modified. Thus, we expect that our algorithm will split `app1history` from the module it has been merged with. Furthermore, we expect that the affected modules `app1avg` and `app1less` are also splitted since they do no longer share input equality with `app2avg` and `app2less`. Figure 6(b) shows that the modification request conducted by Application 1 has been successfully and transparently realized by our algorithm. To conclude our evaluation, our algorithm is able to merge, i.e., optimize, and split module compositions at runtime. The reason why we don't give detailed memory and CPU usage numbers for that particular example is twofold. First, the module compositions are realized in the kernel part of CRAWLER as loadable kernel modules and retrieving such numbers for kernel modules are cumbersome. Second, providing numbers for that particular example is not representative and meaningful: It is easy to construct composition examples with more or less savings.

VII. Related Work

Modular software development process (or modularization) [8], [10], [13], [15] is widely used in the scope networking in order to deal with this rising complexity in designing, implementing and maintaining protocols for distributed systems. Modularization has also be proved to be very practical in cross-layer interaction domain [1]. But in spite of that problems occurring with multiple cross-layer interaction is a very unexplored field. Only the problem of conflicts, i.e., possible performance degradations [9] caused by multiple, contradicting interactions have been mentioned. To the best of our knowledge, the problem of redundancy in multiple cross-layer interactions have not been addressed so far.

However, abstracting the problem of redundancy in cross-layer interactions to redundancy in module compositions the problem can be **theoretically** considered as using directed and labeled graphs. Here, a module is mapped to a node, while a data flow is mapped to a directed edge from the generator to the receiver. This allows us to research a suitable algorithm in the field of graph theory. While there are many interesting findings in the field of subgraph isomorphism, research on the *isomorphic subgraph problem* seems rather scarce. While the isomorphic subgraph problem cannot easily be reduced to other better known graph isomorphism problems, in [3] it is proven to be $\mathcal{NP}$-hard by reduction to the 3-partition problem. If we consider circle free graphs, our algorithm is able to provide an optimal solution in polynomial runtime. Nonetheless, if circularities occur, we may miss optimization possibilities but provide a suboptimal solution in polynomial time. All in all, the need for a polynomial time algorithm depends on the use case: (i) If modules and their compositions are regularly (un)loaded or changed (ii) how big the size of the modules and their compositions are, and (iii) how fast the system needs an optimization. Thus, it may be appropriate to use a variation of the isomorphic subgraph problem. However, since we provide a polynomial time algorithm, our solution will fit to all of these three uses.

Practically, in the field of runtime (self)adaptive software there exists approaches which allow the exchange and modification of modules at runtime [11], [12]. But, these approaches mostly use course granular modules such as in Eclipse, .net and the OSGi framework making it difficult to use our approach since the redundancies in such system will appear less. Another solutions focus more on ontology-based systems [6]



(a) Resulting composition after the merging with our algorithm. Red border shows merged module compositions.

(b) The History-module is exchanged at runtime by another new History-module (red border) causing to split effected modules.

Fig. 6. Merging equal module compositions and splitting them if necessary, i.e., if changes are conducted at runtime to modules and to their compositions.

which requires the developer to put effort to describe the semantic of their modules. In our approach the module developer has to implement once what she considerers as state equality, but from then on due to the reusable nature of the modules, module composition developer can freely compose without putting additional effort.

To summarize, we focus on an automated system that is able to resolve redundancy of multiple cross-layer interactions at runtime and does not burden the module composition developer about finding the overall optimal module composition.

VIII. Conclusion

In this paper, we propose a graph-based redundancy removal algorithm to automatically detect and resolve redundancies in multiple cross-layer interactions. In particular, we used the cross-layer architecture CRAWLER that utilizes module compositions to realize cross-layer interactions. However, we first provide a general theoretical graph-based description of the problem, making it applicable for a wide range of modular systems or networking scenarios. Based on that, we suggest a general algorithm to automatically find redundant module compositions (i.e., parts of cross-layer interactions) and merge them together. With regard to runtime adaptable module compositions we show that more adaption then only a rash removal of redundancies is necessary, since runtime adaptation can lead to invalid module compositions and accordingly suggest how to resolve this issue by bookkeeping. We validate the practical applicability of our approach by implementing it for cross-layer architecture CRAWLER. Our evaluation demonstrates a real use-case where we successfully resolve redundancies in cross-layer interactions at runtime and recreate the original state if necessary.

Although we have applied this generic approach for the cross-layer design domain, the problem can also be mapped to other scenarios such as on to a graph of nodes in the network (wired and wireless networks) to detect redundant node compositions, services, and interactions. Detecting such redundancies can help in turning off services and even nodes, e.g., to save energy in battery driven devices and improve network life time. With our generic solution the adaptation to other fields should be simplified.

IX. Acknowledgments

This research was funded in part by the DFG Cluster of Excellence on Ultra High-Speed Mobile Information and Communication (UMIC), and the DFG special research field MAKI.

References

[1] I. Aktas, F. Schmidt, M. Alizai, T. Druner, and K. Wehrle, "Crawler: An experimentation platform for system monitoring and cross-layer-coordination," in *WoWMoM, 2012 IEEE Int. Symp.* IEEE, 2012.

[2] I. Aktas, F. Schmidt, E. Weingärtner, C. Schnelke, and K. Wehrle, "An adaptive codec switching scheme for sip-based voip," *Internet of Things, Smart Spaces, and Next Generation Networking*, pp. 347–358, 2012.

[3] S. Bachl and F.-J. Brandenburg, "Computing and drawing isomorphic subgraphs," in *Graph Drawing*, ser. LNCS. Springer Berlin / Heidelberg, 2002, vol. 2528, pp. 74–85.

[4] P. Godefroid and S. Khurshid, "Exploring very large state spaces using genetic algorithms," *TACAS*, pp. 71–82, 2002.

[5] T. Goff, J. Moronski, D. Phatak, and V. Gupta, "Freeze-tcp: a true end-to-end tcp enhancement mechanism for mobile environments," in *INFOCOM, 2000. IEEE*, vol. 3, mar 2000, pp. 1537 –1545 vol.3.

[6] S. Götz, C. Wilke, S. Cech, and U. Aßmann, "Runtime variability management for energy-efficient software by contract negotiation," in *Proceedings of the International Workshop on Models@ run. time*, 2011.

[7] J. Hromkovic, *Theoretical Computer Science : Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography*. Springer, November 2003.

[8] N. Hutchinson and L. Peterson, "The x-kernel: An architecture for implementing network protocols," *Software Engineering, IEEE Transactions on*, vol. 17, no. 1, pp. 64–76, 1991.

[9] V. Kawadia, P. Kumar, B. Technol, and M. Cambridge, "A cautionary perspective on cross-layer design," *IEEE Wireless Communications*, vol. 12, no. 1, pp. 3–11, 2005.

[10] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[11] H. Pham, J. Paluska, U. Saif, C. Stawarz, C. Terman, and S. Ward, "A dynamic platform for run-time adaptation," *Pervasive and Mobile Computing*, vol. 5, no. 6, pp. 676–696, 2009.

[12] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

[13] M. Schinnenburg, R. Pabst, K. Klagges, and B. Walke, "A software architecture for modular implementation of adaptive protocol stacks," in *MMBnet Workshop 2007*, Hamburg, Germany, Sep 2007, pp. 94–103.

[14] V. Srivastava and M. Motani, "Cross-layer design: a survey and the road ahead," *Communications Magazine, IEEE*, no. 12, pp. 112–119, 2005.

[15] K. Wehrle, "An open architecture for evaluating arbitrary quality of service mechanisms in software routers," *Networking—ICN 2001*, pp. 117–126, 2001.