

An Approach to Integration Testing Based on Architectural Descriptions

Antonia Bertolino
Consiglio Nazionale delle Ricerche
Istituto di Elaborazione della Informazione
Via S. Maria 46, 56126 Pisa, Italy
bertolino@iei.pi.cnr.it

Paola Inverardi, Henry Muccini, Andrea Rosetti
Universita' di L'Aquila
Dipartimento di Matematica Pura ed Applicata
Via Vetoio, Coppito, 67010 L'Aquila, Italy
inverardi@univaq.it

Abstract

Software architectures can play a role in improving the testing process of complex systems. In particular, descriptions of the software architecture can be useful to drive integration testing, since they supply information about how the software is structured in parts and how those parts (are expected to) interact. We propose to use formal architectural descriptions to model the "interesting" behaviour of the system. This model is at a right level of abstraction to be used as a formal base on which integration test strategies can be devised. Starting from a formal description of the software architecture (given in the CHAM formalism), we first derive a graph of all the possible behaviours of the system in terms of the interactions between the components. This graph contains altogether the information we need for the planning of integration testing. On this comprehensive model, we then identify a suitable set of reduced graphs, each highlighting specific architectural properties of the system. These reduced graphs can be used for the generation of integration tests according to a coverage strategy, analogously to what happens with the control and data flow graphs in unit testing.

1 Introduction

The testing of large and complex systems is a highly difficult and expensive activity whose importance in software development and maintenance cannot be underestimated.

In the system testing stage, we check the overall behaviour of the system against the functional and the performance requirements. System test is the final validation step, and is performed once the whole system is developed. Before it, as design and coding proceed, we test the system in pieces, according to a defined integration test strategy. The objective of a systematic test strategy during development is to detect design failures as early as possible so to arrive at the system test phase with a mature, well functioning system.

Therefore, the individual modules are first tested in isolation, generally by the same developers, in order to early find defects in their implementations. This is the unit test stage. Many white-box and black-box methods now exist for the selection of unit tests [1, 2, 3].

Knowing that the single modules work well in isolation is not enough. Then, we want to test the interactions between them: we merge modules into progressively larger subsystems and select tests to detect possible problems in their interfaces and communications. In this process, there are two key considerations [13]: how to select an effective set of test cases, and how to progressively combine modules in subsystems. In the literature (e.g., [13]), integration testing is mostly addressed with reference to this second issue. Systematic application of top-down, bottom-up, and several other mixed strategies is traditionally recommended, as opposed to chaotic, all-at-once, *big-bang* testing. However, such strategies merely rely on the call structure of the system, i.e., which module calls which other. The description used to specify this call structure is pictorial and quite informal: modules are represented by square boxes, and edges between them represent call relationships. Moreover, only the static aspects of the interactions between modules are considered.

In recent years, the organization of software systems in major design components and the interactions between these components have attracted considerable attention within the software engineering community. A considerable body of work already exists, forming the emerging research field of *Software Architectures* [9, 8, 7, 20, 18]. The software architecture (SA) of a large complex system supplies information about how the software is structured in parts (components of the architecture) and how those parts interact (connectors of the architecture). SAs exhibit both static and dynamic features, have to be described at a high level of abstraction, and must be comprehensible to a variety of users with different educational background. As the complexity of software systems increases, the importance of SAs becomes evident for the analysis, design and construction of the overall system

structure.

Our specific interest in the SA discipline here is in the test and analysis of SAs. In this area, several key challenges face researchers [19], including:

-) how does one test if a design and implementation comply with the SA?
-) how does one specify a SA such that one or more properties can “easily” be tested or analysed?

We claim that addressing such issues amounts exactly at looking for ways to improve currently ad-hoc integration test strategies. In other words, we intend to investigate how integration testing can be planned and controlled based on the SA. A SA description (SAD) provides a model at a right level of abstraction to be used as a formal base on which integration testing strategies can be devised. Indeed, since a SA provides an explicit description of both the static and the dynamic properties of a system, we claim that SADs constitute an important, if not essential, support towards making integration testing a more systematic and automatable process.

We further discuss the possible approaches to and the advantages of *architecture-based software testing* in the next section. In Section 3 we then present a method we are currently working on to derive and control integration tests. Starting from a formal description of the SA, we first derive a state transition graph of all the possible behaviours of the system in terms of the interactions between the components. This graph contains altogether the information we need for the planning of integration testing. On this comprehensive model, we then identify a suitable set of reduced graphs; each subgraph highlights specific architectural properties of the system. Our proposal is to base a formal integration test method on these reduced graphs, analogously to how unit test methods rely on the control and data flow graphs. The graph we derive from the SAD is a finite state automaton; using such models as a base for deriving a design test cases had already been suggested as far as 20 years ago [5]. However, the problem is clearly to manage the explosion of states in the description of large, complex systems. We believe that SAs can today help to control this problem, because they describe the system structure at an appropriate level of abstraction, discriminating among details that have to be included in the description and those that have to be excluded.

Our description of SAs is based on the Chemical Abstract Machine (CHAM) formalism, introduced in [10] and briefly summarised here in Section 3.1. Finally, we draw our conclusions and outline future work in Section 4.

2 Architecture-based integration testing

Traditionally, the testing of large software systems proceeds in subsequent steps, namely unit, integration and sys-

tem test, which correspond in bottom-up order to the code, design and specification stages of the development part of the life-cycle (proceeding conversely in top-down fashion). This approach, descending directly from the functional decomposition principle, allows testers to master the inherent complexity of the task, by subdividing the testing activity into smaller, more manageable pieces, and also enforces a rigorous, disciplined procedure. However, in this approach, formalising and automating the integration test stage is difficult, because the selection of the test cases for the subsystems and the order in which the components are incrementally combined is entirely dependent on the adopted system functionality decomposition, which is normally devised ad hoc.

Moreover, the traditional approaches of picking the components in top-down order, or bottom-up, or their combinations, stress structure over behavior. In fact, it is explicitly stated that the goal of integration testing is to check the compatibility between component interfaces. However, software components naturally embody assumptions about both the structure and the behavior of the contexts in which they are integrated. Besides to conflicts between the information of static type present in the interfaces, when components are put to work together conflicts can also arise in their dynamic interaction [6]. Conventional integration testing, by stressing the structural aspects of the decomposition, is not well suited to uncovering the latter.

Our current research activity is directed to address the issues of integration testing by using the SAD of a system as the primary reference. Indeed, SAs provide a formal and explicit description of both the static and the dynamic properties of a system, as composed by components and connectors.

We claim that this description can be used by the tester to identify an incremental testing plan of the system. A method we are working on for doing this is presented in the next section. Essentially, the approach would belong to the black box techniques of state transition testing, i.e., the system specification is modeled by an automaton, and the generation of test cases is aimed at covering the arcs (i.e., the transitions) and the nodes (i.e., the states) of it. One test case corresponds to a sequence of one or more transitions between states, yielding more or less stringent coverage measures.

The advantages of using the SAD to derive the automaton for state transition testing are evident: first, we have a formal description (in our case a CHAM) and the automaton can be automatically derived; second, the SA is at a high level of abstraction, thus the number of states in the automaton can be kept manageable; third, we can trace changes in the SA to the corresponding modifications in the automaton, allowing for the testing of new systems obtained after insertion or modification of a component.

Moreover, architectures typically specify information also about extra-functional properties of components and connectors. Therefore it should be possible to exploit the architecture specification to test also for extra-functional requirements. So far this has been done for structural and topological requirements [11]. We aim at identifying and incorporating in the test process other objective, quantitative criteria for evaluating the effectiveness of the test cases.

2.1. Architectural test patterns

However, the most appealing issue of using SADs for integration testing has yet to be said. An important direction of research in SAs is to identify a taxonomy of representative, broadly-used architectural patterns, or *styles*: SADs that characterize families of systems, e.g. [9, 18], pipes and filters, object-oriented systems, layered systems, and so on. The idea is that the components and the connectors for a specific style are selected by the software architect from a restricted (to the style) set, and that the ways in which those components and connectors can be combined are constrained by the adopted architectural style.

Designing by “standardized” patterns would have a positive impact for testing, as well. We expect that the complexity of the testing task would be highly reduced because restricted to the style.

Our long term research goal is thus to define (a schema of) an integration test plan for classes of systems that share the same patterns in the way components are structured and interacts. Similarly and in combination to architectural design patterns, this would lead to the derivation of *architectural test patterns*.

3 A formal strategy to integration testing

In this section we describe the method we are currently working on to drive and plan integration testing. Our method is based on the CHAM description of a software architecture, briefly described below.

3.1. CHAM descriptions of software architectures

In the following we briefly summarize the most relevant concepts the CHAM model is based on (see [10] for further details).

A Chemical Abstract Machine is specified by defining *molecules* m, m', \dots as terms of a syntactic algebra that derive from a set of constants and a set of operations and *solutions* S, S', \dots of molecules. Molecules constitute the basic elements of a CHAM, while solutions are multisets of molecules interpreted as defining the *states* of a CHAM. A CHAM specification contains *transformation rules* T, T', \dots that define a *transformation relation*

$S \longrightarrow S'$ dictating the way solutions can evolve (i.e., states can change) in the CHAM.

The transformation rules can be of two kinds: (four) general *laws* that are valid for all CHAMs and specific *rules* that depend on the particular CHAM being specified.

At any given point, a CHAM can apply as many rules as possible to a solution, provided that their premises do not conflict—that is, no molecule is involved in more than one rule. In this way it is possible to model parallel behaviors by performing parallel transformations. When more than one rule can apply to the same molecule or set of molecules then we have nondeterminism, in which case the CHAM makes a nondeterministic choice as to which transformation to perform.

A CHAM description of a software architecture consists of a syntactic description of the static components of the architecture (the *molecules*), a solution representing the initial state of the architecture, and of a set of reaction rules, beside the general ones, which describe how the system dynamically evolves through reaction steps.

The syntactic description of the components is given by an algebra of molecules. Following Perry and Wolf [16], we distinguish three classes of components: data elements, processing elements, and connecting elements. The processing elements are those components that perform the transformations on the data elements, while the data elements are those that contain the information that is used and transformed. The connecting elements are the “glue” that holds the different pieces of the architecture together. For example, the elements involved in effecting communication among components are considered connecting elements. This classification is reflected in the syntax, as appropriate.

The initial solution is a subset of all possible molecules that can be constructed using the syntax. It corresponds to the initial, static configuration of the system. Transformation rules applied to the initial solution define how the system dynamically evolves from its initial configuration.

The CHAM formalism allows for two, quite different analysis techniques which permit to analyze and verify various properties of the system under specification. On the one hand, we can exploit the algebraic and equational nature of CHAM. On the other hand, we can take advantage of its operational flavor to derive transition systems from the specification and then reason at the transition-system level. Here, we are specifically interested in using the transition system for the derivation of a state transition coverage strategy, suitable for incremental testing, as described in Section 3.3.

3.2. The example

We illustrate the method by means of a simple example, the sequential compiler taken from [10] (see Figure 1).

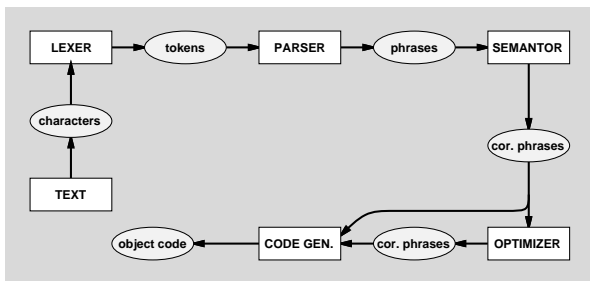


Figure 1. Sequential multiphase compiler architecture.

The syntax Σ_{seq} of molecules M in the sequential architecture is:

$$M ::= P \mid C \mid M \diamond M$$

$$P ::= \text{text} \mid \text{lexer} \mid \text{parser} \mid \text{semantor} \mid \text{optimizer} \mid \text{generator}$$

$$D ::= \text{char} \mid \text{tok} \mid \text{phr} \mid \text{cophr} \mid \text{obj}$$

$$C ::= i(D) \mid o(D)$$

where P is the set of processing elements, D is a set of constants representing the data elements and C of connecting elements of the compiler. In this case, the connecting elements consist of two operations, i and o , acting on the data elements and representing the communication ports, input and output respectively. At the molecules level, we also introduce an infix operator “ \diamond ” that we use to express the status of a processing element with respect to its input/output behavior. For example, the molecule $i(\text{tok}) \diamond o(\text{phr}) \diamond \text{parser}$ represents the parser in the state in which it can consume tokens as input and then go in a state in which it can produce phrases as output. As we said in the previous section, a state change is achieved by applying suitable reaction rules.

We take as the set of syntactic elements the initial algebra in the class of all the Σ_{seq} algebras.

The next step is to define an initial solution S_1 . This solution is a subset of all possible molecules that can be constructed under Σ_{seq} and corresponds to the initial, static configuration of a system conforming to the architecture. Transformation rules applied to the solution define how the system dynamically evolves from its initial configuration.

$$\begin{aligned} S_1 = & \text{text} \diamond o(\text{char}), \\ & i(\text{char}) \diamond o(\text{tok}) \diamond \text{lexer}, \\ & i(\text{tok}) \diamond o(\text{phr}) \diamond \text{parser}, \\ & i(\text{phr}) \diamond o(\text{cophr}) \diamond \text{semantor}, \\ & i(\text{cophr}) \diamond o(\text{cophr}) \diamond \text{optimizer}, \\ & i(\text{cophr}) \diamond o(\text{obj}) \diamond \text{generator} \end{aligned}$$

The final step is to define three simple transformation rules.

$$T_1 \equiv \text{text} \diamond o(\text{char}) \longrightarrow o(\text{char}) \diamond \text{text}$$

$$T_2 \equiv i(d) \diamond m_1, o(d) \diamond m_2 \longrightarrow m_1 \diamond i(d), m_2 \diamond o(d)$$

$$\begin{aligned} T_3 \equiv & o(\text{obj}) \diamond \text{generator} \diamond i(\text{cophr}) \longrightarrow \\ & i(\text{char}) \diamond o(\text{tok}) \diamond \text{lexer}, \\ & i(\text{tok}) \diamond o(\text{phr}) \diamond \text{parser}, \\ & i(\text{phr}) \diamond o(\text{cophr}) \diamond \text{semantor}, \\ & i(\text{cophr}) \diamond o(\text{cophr}) \diamond \text{optimizer}, \\ & i(\text{cophr}) \diamond o(\text{obj}) \diamond \text{generator} \end{aligned}$$

where $m_1, m_2 \in M$ and $d \in D$. The first rule makes source text available for compiling. The second rule is a reaction rule that describes pairwise input/output communication between processing elements. In particular, communication occurs if there is a processing element m_1 that consumes input d produced as output by some other processing element m_2 . Completion of the communication—that is, the result of the transformation—is indicated by a rewriting of the molecule such that the data port is moved to the rightmost position of the molecule. The third transformation rule restores the processing elements to their initial states after the code generator has completed its task and offered object code as output. This allows the system to begin processing a new source text.

As already said, from this description we can derive a transition system. Basically, we use the usual approach of deriving the transition system from the operational semantics [14] by considering that our reaction rules are indeed the operational semantic rules. This derivation process can be automated and indeed we have a tool which takes as input a CHAM description [17] and generates a transition system. The transition graph for the sequential multiphase compiler is given in Fig. 2.

Note that, due to the concurrent nature of the CHAM formalism, in the graph there are also arcs that represent the concurrent application on the same solution of more than one reaction rule. Note also that transition graphs are labelled on arcs with the labels of the reaction rules whose application has determined the transition. This piece of information will be used when reducing the complete graph with respect to specific views. Actually we keep also the information associated to nodes, that is the solutions, since this is useful in the generation of a particular kind of subgraph as detailed in the next section.

3.3. The method

The incremental test method we propose is based on the derivation of suitable subgraphs of the CHAM-derived transition graph. These subgraphs represent specific views

of the system behaviours. The idea is to use these views as a base for the definition of coverage criteria and testing strategies.

The methodology is to derive a specific subgraph by selecting an appropriate criterion. For example one criterion could be to take all the derivations containing the application of a given rule. Then the subgraph thus obtained can have to be completed with respect to the complete one, in order to obtain a connected subgraph. In this way all the subgraphs that we define are connected subgraphs of the original transition system.

So far we have defined some criteria, according to which different subgraphs are obtained. The idea is that for each subgraph one derives a test plan including enough test sequences to cover all the transitions in these subgraphs. Clearly, one may want to test more different subgraphs, that is the SA is tested according to different criteria. For a complete theory with the description of the algorithms to derive the subgraphs see [17, 15, 4].

In the following we only show and discuss the concurrent-reduced graph (see Fig. 3), the input-reduced graph (Fig. 4) and the regression-reduced graph (Fig. 5) with respect to the Lexer component of the sequential multiphase compiler.

Concurrent-reduced graph: This graph is obtained according to the following rule: if more paths reach a given state and one among them is a concurrent arc, only the latter is kept. The rationale under this view of the system is that of providing a sort of minimal set of the complete behaviours that have to be considered. In fact, we have selected the concurrent arcs and deleted the (interleaving) ones since whatever scheduling policy the system implementation will adopt, it will certainly have to consider at least this subgraph. Looking at the picture note that the only concurrent arc that is kept is the one connecting S_2 to S_5 . All the other concurrent arcs do not appear in the concurrent graph since they originate from node S_4 which is only reachable through the interleaving arc labelled T_2 leaving from S_2 and that is deleted when keeping the concurrent arc from S_2 to S_5 .

Input-reduced graph: The input-reduced graph considers all the paths that can be reached when a finite input is given, i.e., we consider the behaviour of the system when a finite input from the external world is provided. In our example the external input is represented by the Text. Figure 4 represents the input-reduced graph for only one input. It means that we are tracing all the possible computations in the graph that had origin by a single application of the rule which provides the input, in this case T_1 . The same process can be applied by considering 2 or more applications of rule T_1 , thus obtaining larger and larger graphs. This process can be iterated until the complete graph is obtained. The number of inputs (occurrences of the input rules) considered in the last iteration represent the maximum amount of *concurrent* input that can be managed by the system. In a certain sense

this subgraph allows for a *stress* testing strategy.

Regression-reduced graph: it considers only that portion of the complete graph dealing with the behaviour of a finite number of processes, e.g. just one. The regression-reduced graph highlights only that portion of the graph which describes the behaviour of the system with specific reference to a given component. In Fig. 5 this is shown with respect to the Lexer component. The algorithm that generates this graph not only uses the information associated to arcs but also uses the information associated to nodes, that is the solutions. It is in fact necessary to analyze all the solutions in order to identify all the states that involve the considered components. This kind of graph can provide a useful base for the definition of *regression* testing strategies.

Having derived these and possibly other reduced graphs, we have then to select test cases covering them according to a selected coverage criterion. Similarly to control flow and data flow coverage for unit testing, more or less stringent coverage criteria could be adopted here to derive sequences of states covering adequately these subgraphs. In [5], for instance, several transition coverage criteria, and their respective efficacies, are discussed.

The outlined approach of deriving several subgraphs from the complete automaton, according to different reduction criteria, and then selecting tests for each of them is aimed from one hand at controlling the inherent complexity of the state transition graph of a large system. On the other hand, we thus provide a formal and explicit way to test a system according to different criteria. In fact, it is now accepted that there is no test strategy that is the optimal one for the testing of software, but rather a suitable combination of more, different test strategies should be followed. This is reflected in our method in the derivation of different subgraphs, whereby coverage of each corresponds to satisfaction of a different test strategy.

Finally, we believe that by our approach we can find a practical, and rigorous solution to the problem of testing a system made of re-used, or standardised, components. The method we intend to pursue is that above outlined of the regression-reduced graph. From the SA transition graph we derive the subgraph of interest for testing the system after the insertion of a component taken for instance from a library.

4 Conclusions and future research

We are working at developing a rigorous, automatable method for the integration testing of complex systems, based on the formal description of the software architecture. The key idea of this method is that, starting from the transition system of the SA automatically derived from its formal specification (here in CHAM), we derive one or more reduced subgraphs, each providing a partial view of the SA, suitable for the testing of specific aspects of the system behaviour.

On these subgraphs we then derive a set of test sequences aimed at covering the state transitions.

In this paper we have only provided a preliminary outlook of the method. However, we believe that this is enough to show that the method and the tools we intend to exploit are sound and promising.

Future work will be to refine the method and to verify its validity on realistic case studies. We also intend to investigate the feasibility of generalising the method presented in [12] to the coverage testing of the reduced transition graphs. That method has been defined for a family of coverage strategies of the data and control flows of a software unit and consists in identifying the *minimum* subset of coverage items which would guarantee the coverage of the whole set. This minimum set allows for the reduction and the estimation of the cost of testing. We intend to investigate a similar method for coverage testing at the level of the software architecture, i.e., for integration testing. This means that we first specify the test criteria to be satisfied by the tests with reference to our reduced subgraphs, and then we derive the minimum set of transitions to be exercised by the test of the system.

Our long term research objective is twofold:

- provide a rigorous yet practicable procedure for the testing of large, complex systems. Starting from the formal specification of the SA, we intend to provide a well defined strategy which produces an efficacious test plan.
- provide a set of architectural test patterns. We will consider some specific architectural styles and will systematically apply to them our method in order to derive patterns of integration test sets, which can then directly instantiated by the testers to architectures belonging to the considered style.

References

- [1] M. A. Adrion, M. A. Branstad, J. C. Cherniavsky Validation, Verification and Testing of Computer Software. *ACM Computing Surveys*, 14(2):159–192, June 1982.
- [2] B. Beizer. *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, New York, 1990.
- [3] B. Beizer. *Black-Box Testing Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., New York, 1979.
- [4] A. Bertolino, P. Inverardi, H. Muccini, A. Rosetti. A Formal Approach to Architecture-based Integration Testing. Technical Report University of L’Aquila, in preparation.
- [5] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4 (3), May 1978.
- [6] D. Compare, P. Inverardi and A. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. Internal Report, University of L’Aquila, June 1996.
- [7] D. Garlan Editor. ICSE17–Workshop on Software Architectures. In *Proceedings of the 17th International Conference on Software Engineering*. IEEE Computer Society, April 1995.
- [8] D. Garlan and D.E. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.
- [9] D. Garlan and M. Shaw. An Introduction on Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*. World Scientific, 1993.
- [10] P. Inverardi and A.L. Wolf. Formal Specifications and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):100–114, April 1995.
- [11] P. Inverardi and D. Yankelevich. Relating CHAM descriptions of Software Architectures. *IEEE Proc. IWSSD-8*, March 1996, Paderborn.
- [12] M. Marré and A. Bertolino. Reducing and estimating the cost of test coverage criteria. *Proc. ACM/IEEE Int. Conf. Software Eng. ICSE–18*, pp. 486–494, Berlin, Germany, March 1996.
- [13] G. J. Meyers. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, 1979.
- [14] R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
- [15] H. Muccini. Test di Integrazione basato su specifiche architetturali Tesi di Laurea , Università’ di L’Aquila, Marzo 1997.
- [16] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.
- [17] A. Rosetti. Analisi e Testing di Architetture Software descritte tramite CHAM |newblock Tesi di Laurea , Università’ di L’Aquila, Marzo 1997.
- [18] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice All, 1996

- [19] W. Tracz. Test and Analysis of Software Architectures. *Proc. of the ACM International Symposium on Software Testing and Analysis (ISSTA'96)*, January 1996, S. Diego.
- [20] L. Vidal, A. Finkelstein, G. Spanoudakis, A. Wolf, Editors. Second International Software Architecture Workshop ISAW-2. In *Joint Proceedings of the ACM SIFSOFT '96 Workshops*. ACM, October 1996.

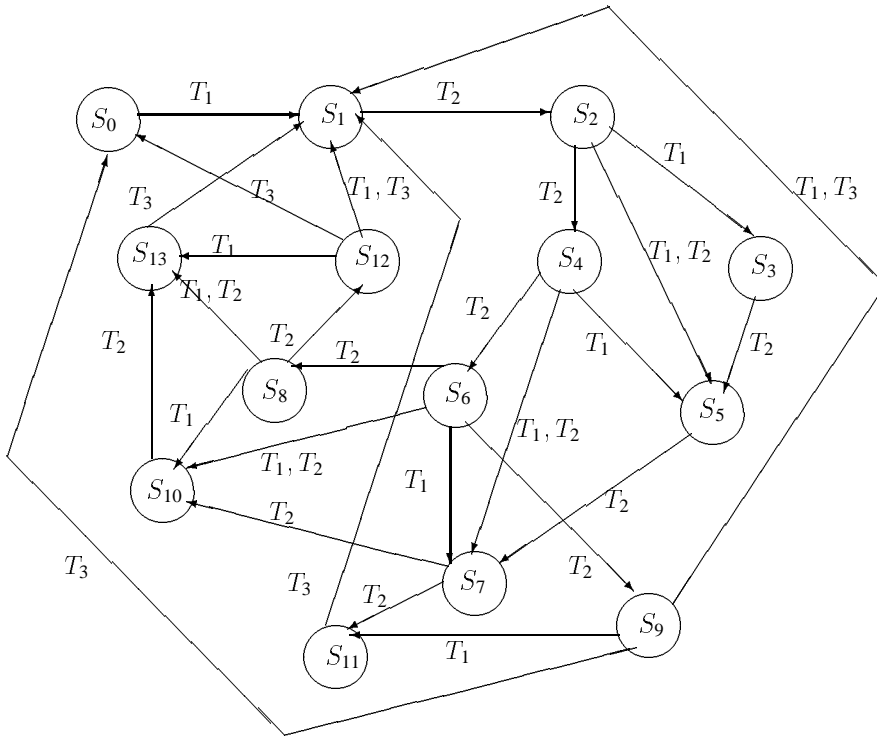


Figure 2. Transition graph for the sequential compiler.

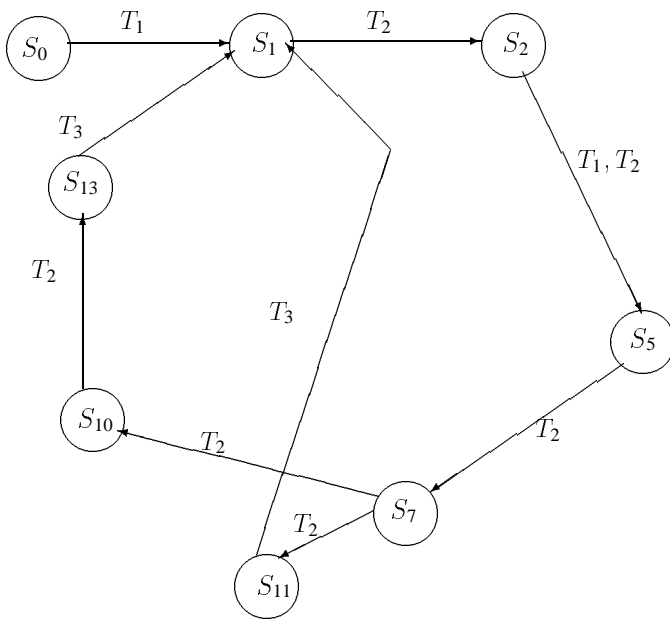


Figure 3. Concurrent-reduced graph.

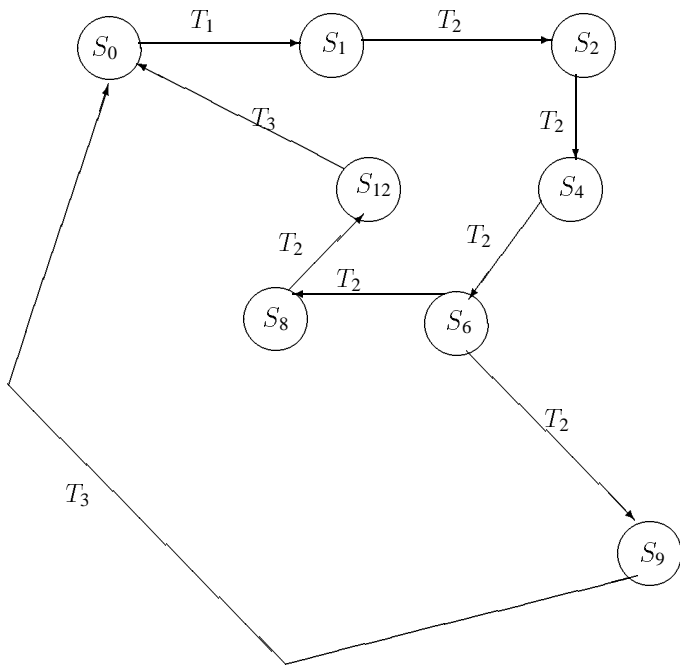


Figure 4. Input-reduced graph with respect to input rule T_1 .

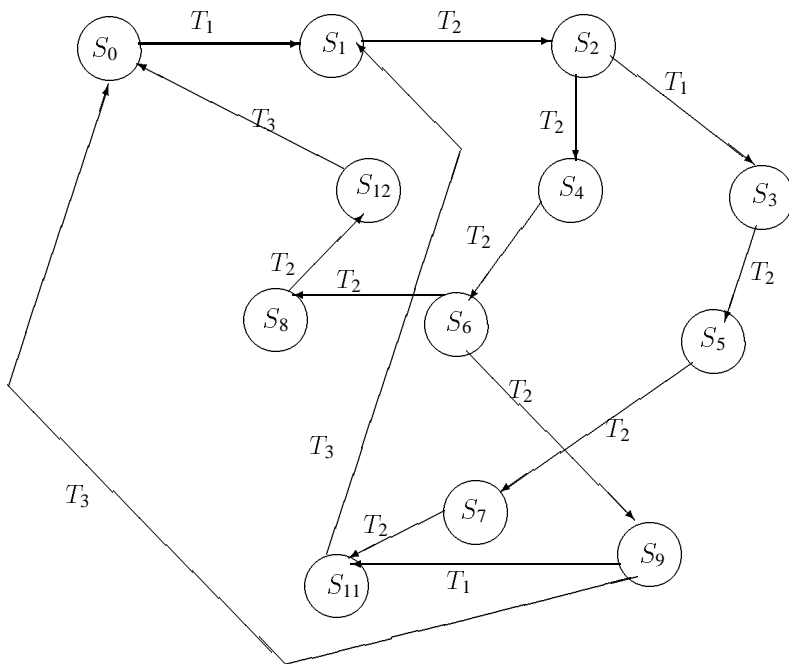


Figure 5. Regression-reduced graph.