A Component-Based Software Engineering Approach for Developing Trustworthy Systems

Mubarak Mohammad
email: ms_moham@cse.concordia.ca

Vasu Alagar
email: alagar@cse.concordia.ca
ABSTRACT

Developing trustworthy software systems that are complex, and used by a large heterogenous population of users is a challenging task. Component-based software engineering (CBSE) has many attractive features that can provide an effective solution to these challenging issues. However, the essential requirements of CBSE have not been met in the current approaches. Therefore, we present a CBSE approach that involves three contributions. The first contribution is a component model that defines the trustworthiness quality attributes as first class structural elements. This enables us to formally verify trustworthiness properties and demonstrate that a high level of trustworthiness has been achieved. In our approach, formalism is integrated into the various stages of the development process. So, our second contribution is a process model that plays this role. The third and final contribution is a development framework of comprehensive tool support. We describe the tools and justify their role in assuring trustworthiness during the different stages of software development.
1 Introduction

This paper is a contribution to component-based software engineering (CBSE). It presents a systematic approach for the development of trustworthy component-based systems. The approach comprises a trustworthy component model, a development process model, and a framework of comprehensive tool support that implements the development process model. In this approach, rigor is applied throughout the entire system life cycle.

Setting the stage: Software systems are increasingly becoming ubiquitous affecting the way we experience life and perform work. For example, smart devices and intelligent sensors are currently used to capture information about human activities along with their physiological and psychological status and communicate it through wireless connections [13]. The collected information triggers adaptation in a pervasive environment according to predefined preferences. Such systems are being used in the healthcare sector to improve its services. Another example can be found in avionics. Currently, aircrafts are being controlled fully by autopilot, a computer software that guides the aircraft. These examples show the current advancement of software development in areas that affect our daily lives. At the same time, it raises questions about the ability of the current software development paradigms to cope with the risky trends of pervasive computing. In turn, pervasiveness raises concerns about trustworthiness: to which extent the current software development paradigms are capable of producing trustworthy systems that control the lives of people and manage their private data?

What is trustworthiness? Trust is a social aspect that is hard to define formally. Trust is relative, there is no absolute trust. We depend on technology on which aircrafts, trains, and elevators serve us in our daily life although it fails from time to time causing catastrophic consequences. Yet, we still use it because it has passed a minimum level of acceptance in providing services. Hence, there is a need to design systems that are provably trustworthy. Towards this purpose, the credentials of trust should be formally defined along with their level of acceptance. Trustworthiness should be defined as the system property that denotes the degree of user confidence that the system will behave as expected [21][7]. In the litera-
ture [21][7], there is a common consensus that essential quality attributes contributing to trustworthiness are availability, reliability, safety, and security. Since many of the current systems are real-time, we also include predictability and timeliness to the quality attributes of trustworthiness. Predictability implies that with certainty, an action is enabled in each system state. Timeliness refers to bounded time constraint behavior. When analyzing these attributes, we find that availability and reliability are quality of service properties that can be analyzed and measured during system execution based on usage profiles. On the other hand, safety and security are design considerations that can be specified and verified during the design activities of system development. This raises questions such as: how these properties can be satisfied collectively in one development process?, and whether the current state of the art of software development paradigms can address their requirements?

**Why CBSE?** CBSE promises many advantages to software development including reuse, managing complexity, and reducing the development’s time, effort, and cost. CBSE is widely adopted in software industry as the mainstream approach to software engineering [22]. It is increasingly used to develop software systems, especially embedded systems, deployed in safety critical environments. Complexity is effectively managed by dividing the problem into subdivisions, each of which handled separately in CBSE. The cost of development is reduced by reusing existing solutions to these subdivisions. Therefore, can CBSE be used to develop systems which are pervasive and trustworthy? In order to answer this question, we first investigate whether or not CBSE has fulfilled its initial intended promises.

**Has CBSE met its objectives?** The essential constituents of CBSE are components engineering, component model, and development process model [22][14][11]. In the following, we briefly discuss these elements and motivate why their requirements have not been met in the current approaches of CBSE.

Component engineering addresses the issues of component’s specification, development, qualification, documentation, cataloguing, and adaptation and selection for reuse. In general, software systems implement functional and non-functional requirements. This implies that component specification methods and qualification techniques should support both functional and non-functional requirements.
However, generally, current component-based development approaches have limited or no support for non-functional requirements, see Section 2. Furthermore, non-functional requirements and environmental constraints should be defined as contracts at the interfaces of a component. This is because environmental assumptions and non-functional requirements might not be valid when using components in different contexts. For example, in real-time embedded systems, time constraints that define the maximum amount of time for a safe execution of a service might be different depending on hardware and software configurations of the deployment environment. The separation between the computation part of the component and its contract enables components to be reused in different environments by changing only its contract. However, when studying current component models, there is limited or no support for contracts. Therefore, current component engineering practices can only support limited reusability of components.

A component model addresses the issues of assembling components to develop component-based systems. There are two major concerns when assembling components: encapsulation and non-functional properties. First, components should be self contained and loosely coupled. The composition mechanism should preserve encapsulation of component’s data and control. This is done by separating the computation part of the component from its interactional specification. A study of current component models [16] revealed that components are composed using direct method calls or indirect message passing through connectors. Thus, these models produce tightly coupled components that are difficult to reuse. Second, when assembling components, special composition rules should be applied to ensure that the non-functional requirements of the constituent components are preserved in the assembly. This requires a defined composition theory. However, there is no component model that defines a composition theory for both structural and nonfunctional parts of a component [16].

A development process model addresses the activities involved throughout the entire component and system lifecycle. CBSE comprises two parallel activities: software component development and component-based systems development (CBD) [12] [20]. The former concerns about component engineering and the later concerns about component models. The main focus of current component models has been on technology [12]; no process model has been defined for developing trustworthy systems.
The above discussion shows that: (i) current practices of CBSE are not based on rigorous process models, (ii) current component engineering lacks support for non-functional requirements, and (iii) more work needs to be done to achieve practical reusability. This raises a major concern regarding applying the current CBSE practices for the development of safety and security critical systems and pervasive computing.

Our contributions It is in this context that we provide a three-fold contribution to a perspective of CBSE: (i) a formal component model that collectively addresses the requirements of trustworthiness, reusability, and CBD, (ii) a formal development process model that describes component engineering and CBD, and (iii) a framework with a comprehensive set of tools that support the formal development process.

The rest of this paper is organized as follows: Section 2 presents a brief analysis of current component models focusing on their support for trustworthiness attributes. Then, we highlight the distinctive features of the approach presented in this paper. The new component model is presented in Section 3. The new process model is introduced in Section 4. In Section 5, the development framework is introduced. It presents the various tools used throughout the development process. The tools are discussed and compared with the most relevant recent works in CBD tools. Finally, Section 6 concludes the paper.

2 An Analysis of Current Component Models

This section briefly compares the existing component models. The models are SOFA 2.0 [10], Fractal (there is variety of Fractal implementations, in this paper we refer to [8]), KobrA [6], Koala [24], CORBA CCM [1], PECOS [19], and Pin [15]. We add SaveCCM [3] to this list. A detailed comparison and discussion of these component models is presented in [16]. In this paper, we are concerned only with their support for non-functional requirements, component composition, real-time requirements, and protocol analysis. The rational behind selecting these component models is that: (i) these models have implemented some of the requirements of CBSE and they have some tools supporting them, and (ii) these models have been considered as component models in a taxonomy of software component
models [16]. After studying these component models we have come up with the following findings:

- **Non-functional Requirements**: PECOS and SaveCCM allow the inclusion of temporal requirements as non-functional requirements in the definition of components. All other component models do not provide any support for including non-functional requirements in component definitions.

- **Component Composition**: None of the current component models have a compositional theory for both structural and non-functional requirements [16].

- **Real-Time**: Only PECOS, Pin, and SaveCCM are real-time component models that support real-time analysis at design time using formal verification.

- **Protocol Analysis**: SaveCCM provides tools to perform design time analysis using simulation and formal verification. Both SOFA and Fractal support analysis at all development phases for protocol compliance. The analysis ensures that components behave as defined by their behavior protocol. However, their current compliance analysis at implementation and run-time can handle only non-parameterized protocols. All other component models provide no support for implementation and run-time analysis of either functional or non-functional requirements.

This brief analysis justifies our early remark that the current approaches of CBSE don’t have the essential requirements to support developing trustworthy systems. Therefore, there is a need for a new perspective in defining components and assembling them to create component-based systems that supports non-functional requirements, specifically, those related to trustworthiness. The CBSE approach proposed in this paper has the following distinctive features:

- **Tools support**: Tools support is an essential factor for achieving consistent results. Specialized tools are included to tackle the various activities of components and system development. Detail discussion of these tools is provided in Section 5.

- **Contract definition**: The component’s formal model clearly distinguishes and strongly couples the structure and contract parts of the component specifications. The security, safety, liveness, and timeliness requirements are defined
in the contract part of the component definition. This separation maintains encapsulation and allows reuse of components in different contexts.

- **Early analysis:** Analytical techniques can be applied at an early stage of component and system development. This is because the contract definition is associate with the structure part of a component. This enables reasoning about trustworthiness at the structural level.

- **Composition:** The composition mechanism is specified in the contract. Therefore, components are self contained and loosely coupled. Composition is supported both at design time and deployment time. Design time composition is done during the component development process. Deployment time composition is done during system development process.

- **Behavior protocols:** The approach supports different protocols for describing the behavior of trustworthy components. Based on defined formal transformation rules, a compiler tool can automatically generate different types of behavior protocols.

- **Support for different verification tools:** The presented approach supports different verification tools in order to verify a wide range of properties and target different kinds of systems. This is necessary because different verification tools differ in the expressive power of their modeling language and verification methods.

- **Support for real-time:** The presented approach supports the development of un-timed systems as well as real-time systems.

### 3 Trustworthy Component Model

This section introduces our perspective of a trustworthy component model. Trustworthiness involves achieving availability, reliability, safety, security, timeliness, and predictability. Availability and reliability are run-time quality of service requirements, whereas the rest are specifiable at design time. Hence, we define safety, security, timeliness, and predictability as the credentials of trustworthiness at component’s structural level. We address methods for achieving availability
and reliability as part of our rigorous development process presented in Sections 4 and 5. The component model includes: (i) the essential structural elements of CBD, such as components, interfaces, and connectors, and (ii) the trustworthiness features. Figure 1 depicts the component model. Below we discuss its contents in details.

![Component Model Diagram](image)

**Figure 1: Component Model**

**Component Definition:** A component is an instance of a *component type*. A component type is an aggregation of *interface types*, which are access points through which *services* are provided and requested. An interface type enumerates the set of services that are available in that interface. A service represents an implementation of a functionality. A service can be offered by only one interface type and it can have multiple *data parameters*. The specification of services and data parameters as first class structural elements is a distinctive feature of this component model. It allows non-functional requirements and constraints to be defined on services and data parameters in the component’s contract.
**Contract Elements:** A contract specifies components’ interactions and non-functional requirements. It consists of *reactivity* specifications, and may optionally include *time constraints*, and *data constraints*. A reactivity specification describes a relation between two components in which one component requests a service and the second component provides a service as a response. Thus, a reactivity specification is a relation between requests for services and their corresponding responses. This controls the interactions between components. Predictability of component behavior is achieved by the formal specification of reactivities. That is, for every request received, its expected responses are defined precisely. Therefore, the internal implementations of services contain no direct or indirect method calls going outside the component because interactions are specified in the contract as reactivities. Therefore, components are loosely coupled and self contained. Reactivity specification can be modified without impacting the implementation of services.

A request for service can have multiple possible responses. A data constraint is a guard condition used to enable a reactivity relation. It can be expressed as a conjunction of predicates over the values of data parameters associated with the request. In case a request has many possible responses, it is possible to define mutually exclusive constraints for responses in order that only one response will be selected, based on the truth value of the defined data constraints. A time constraint defines real-time requirements to regulate services. As an example, “a valve must open within 5 seconds of receiving the request for opening it” is a real-time constraint. Data and time constraints are specified using first order predicate logic. A safety property defines an invariant over the behavior of the component. It regulates the manner in which services are provided or entertained.

Security involves restricting access to services and/or data communicated with services. We view *security mechanism* as a structural element. The only security mechanism that we have included in the first version of our current models is *role based access control*. In principle, the component model is general enough to include many other security features. A *user* represents an identity on whose behalf the components execute. A user can belong to one or more *groups*. A *role* defines responsibilities that can be taken by a user or a group in the system. A role enumerates a set of *privileges*, each of which defines an access permission to perform a service or access the value of a data parameter. The security property is defined as a combination of *service security* and *data security*. Service security aims to
secure the services provided by the component. It states that: (i) *for every request received at the interfaces of a component the request should be received from a user who has permission to request the service*, and (ii) *for every response sent by the component the user who will receive the response should have permission to receive it*. Data security aims to secure the data communicated with services, it states that (i) *for every request received and for every data parameter in the request, the user sending the request should have permission to access the data parameter*, and (ii) *for every response sent and for every data parameter associated with the response, the user receiving the response should have permission to access the data parameter*. If a user doesn’t have proper permission to send a request then the request will be ignored; also, if a user doesn’t have a permission to receive a response, the response will not be sent. In the same manner, if a user doesn’t have permission to access a data parameter, the data parameter value will be set to null value. A component that has no security restriction will respond to every request received by it.

The inclusion of contracts as first class structural elements in component definition is a distinctive feature of our CBSE perspective. It improves the reusability of components. This is because the contract settings can be modified to satisfy different systems and environments without altering the implementation of services.

**Structural Elements:** A component type can be either primitive or composite. A composite component type is obtained by assembling multiple component types connected together using *connector* types. Connectors are defined based on reactivity specification. A connector is defined to link two interface types if there is a reactivity relation between the services declared at the two interface types. One or more *architectures* can be associated with a component type. An architecture is an implementation of the component type associated with it. In an architecture, components initialized from component types are connected by tying their interface types to different connector roles of the same connector. A *connector role* defines an access point for communication.

**System Definition:** A *package* can include definitions of elements explained above to facilitate the reuse of relevant definitions. An *attribute* defines a semantic information that can be assigned to any element using typed-named values. *Constraints*
that are defined as predicates on the attributes can be assigned to any element. System definition includes hardware components and configuration. In the definition of hardware components, specification of deployment units that will host software components are included. In configuration definition, a system specification is included. Typically, this includes descriptions of hardware and software components along with initialization of their local attributes, specification of system users and their assignment to security groups and roles, and deployment specification. A deployment specification states for each software component the hardware component in which it will be deployed. It is possible to define different system configurations for different deployment environments by adjusting attributes and constraints. Hence it promotes reusability.

Defining a new component model is not sufficient to ensure trustworthiness. There is an essential need for a rigorous development process that ensures that trustworthiness is maintained throughout system lifecycle.

4 The Process Model

This section introduces our perspective of a rigorous process model for developing trustworthy systems. The process model lays down the activities in the development process. It integrates formal methods with the phases of the system lifecycle. In particular, it incorporates iterative development, incremental design, validation, formal verification, traceability analysis, and certification. The process model consists of two parallel tracks in which the activities of component engineering can occur concurrently with component-based system development. Figure 2 depicts the two parallel process models. The following explains the two process models in detail.

4.1 Component Process Model

Figure 2(a) shows the component process model. During system design, designers and integrators can reuse existing components. If there exists no component which satisfies the requirements, a new component should be developed. Therefore, the component process model consists of two tracks: component reuse and component development.
Component reuse: System designers start searching for candidate components that could satisfy the stated requirements, both functional and non-functional. We assume a component repository that hosts existing components along with their specification, implementation, and documentation. If the search results in finding some components, the selection task is carried out to qualify the candidate components and select the most appropriate one. Selection is based on domain knowledge from one hand and component meta data retrieved from the repository. The meta data includes some usage profiles, if any, along with component certificates specifying its non-functional qualities. If the component requires some modifications to fit in the new deployment environment, the adaptation task is carried out to perform the required modifications. These modifications must be tested and the code and traceability analysis iterative task is performed to ensure that the required trustworthiness attributes are implemented in the new component. After finishing the analysis, the component is certified and stored in the component repository for future reuse.

Component development: The requirements of new components are used to formally specify them using the component model discussed in Section 3. The formal specifications are validated against defined syntactic and semantic correctness rules in the validation task. After completing the validation, formal verification
is carried out to ensure that the design satisfies the required trustworthiness properties. If the verification task fails, the component is redesigned and the process is repeated until the component design is assured to satisfy the required properties. Then, the implementation task is performed and the implemented component passes through iterative cycles of code inspection and traceability analysis to ensure that the implementation satisfies the verified design. Finally, the new component is certified and stored in the component repository. After having all the required components stored in the component repository, system process model starts to build the component-based system by assembling existing components and verifying that the composite system satisfies the trustworthiness properties.

4.2 System Process Model

Figure 2(b) shows the system process model. It requires a formal model of the environment to be produced and integrated with system elements that are selected from the component repository. The formal specification of the selected software components are composed to define the formal model of the software unit. Environmental components are abstracted and their requirements, constraints, and interfaces to system components are formally defined as a formal model of the environment. System requirements which include functional and non-functional (trustworthiness) requirements are identified and their formal descriptions are produced. A trustworthy system model is composed of the software unit and the model of the environment. A formal model of the deployment plan is defined to describe the relationship between system components, environment components, and deployment units.

The formal model of the trustworthy system is not implemented until several iterations of design take place. First, the formal models are validated for syntactic and semantic correctness. After that, the system design is mechanically translated [4] into a behavioral model on which a formal verification can be conducted. We use model checking techniques. The properties of trustworthiness such as safety, liveness, time constraints, and security are verified formally at this stage. An iterative cycle enables redesigning the system when the design does not satisfy the desired properties. Next, implementation of the software unit is created by retrieving components implementation from the repository and composing them
together. The implementation undergoes an iterative cycle of traceability analysis to ensure that it conforms to the specified formal models and satisfies the defined trustworthiness properties. Then, the software is deployed according to the formal deployment plan in a run-time environment that allows dynamic reconfiguration. While in operation, the run-time software is continuously monitored and analyzed to ensure that its behavior is restricted by the trustworthiness properties and conforms to the verified formal models. Run-time monitoring is a powerful mechanism to ensure availability and analyze reliability.

Defining a rigorous development process model is not sufficient to ensure trustworthiness. It requires tools support to implement the various tasks and ensure their systematic flow.

5 Framework Architecture

The framework is being built on the rigorous process model. Tools are provided to practise CBSE to develop trustworthy systems. The framework can be viewed in three layers: design, implementation, and deployment. Taken as a whole, the framework describes the tools necessary for the different stages outlined in the process model. Figure 3 depicts the framework architecture showing the tools in the three layers. The tools themselves are being developed using our CBSE perspective in a rigorous manner and we strictly follow the process model. Below we give a detailed description of the tools and highlight the merits of each tool.

5.1 Design-time tools:

1- Visual modeling: This tool provides a user friendly interface to model components and systems and specify functional and non-functional properties. It acts as interface to perform design without being directly exposed to the formal notation. The tool projects both textual and visual representations of the design. Also, it projects the model into 3 different views for different users: CBD, real-time, and trustworthiness view. The tool has been implemented using Java. Every architectural element has a defined visual representation. The user designs a system by dragging and dropping visual elements into a design canvas. Relations, properties, attributes, and conditions can be associated with the design elements. The system
specification are saved in an XML file. The specification of system’s elements are transformed from visual representations into XML descriptions according to a defined architectural description language [18].

2- Compiler : This tool checks the syntactic correctness of the visual modeling design with respect to its abstract definitions. The compositional correctness of component design elements and the architectural mismatches such as incompatibility of the interface types defined in the connector types or those used in the architectures of composite components are checked. Error messages are given when inconsistent or incompatible definitions appear in the design. If the design is syntactically correct, the compiler generates a formal descriptions of the visual model. The compiler generates different types of output by transforming the valid design according to formally defined transformation rules. The current version of the compiler generates three types of output: (i) a textual description in an architectural description language [18], behavioral model descriptions in different
formal notations such as UPPAAL [9] timed automata and real-time-Promela [23], and a real-time model using timed automata extended with tasks [5].

Manual transformation of component specification at design time into other models is complex and error-prone. Therefore, applying automatic model transformation techniques is very important to ensure a highly convincing level of trustworthiness. The transformation process is implemented using XSLT [2], a standard mechanism for transforming XML documents into other types of documents. The process uses formally defined transformation rules [4] to transform the system specification from the saved XML file into the required output format. Currently we use a text file specifying ADL or an XML file specifying an extended timed automata. The transformation rules are implemented using XSLT instructions and XPath, an expression language for finding information in an XML document. An extended timed automata is generated by mapping components into templates (an extended timed automata), services into states, reactivity relations into transitions and tasks, data constraints and security specifications into guard conditions, time constraints into state invariants, and attributes such as period and priority into task attributes. The implementation of this tool can be easily extended to accommodate more views. This is because the transformation is implemented using XSLT. This means that the transformation rules can be maintained, updated, and extended without affecting the transformation process or requiring reimplementation.

3- Transformation analysis : Automatic model transformation is increasingly being adopted as a technique to reduce the complexity and faults of transformation. However, the correctness and completeness of the transformation process must be subject to reasoning. Design and implementation flaws are still possible during the development of the automatic transformation tools. Therefore, it is necessary to subject the transformation process to inspection in order to make it trustworthy. The transformation analysis tool is crucial to verify the correctness, completeness, and compatibility of the views produced by the transformation process of the compiler. A view is complete with respect to the visual model if every feature in the view is a feature in the visual model. That is, there is no extraneous feature in a view. A view is correct with respect to a visual model if the view is complete and every feature in the visual model is mapped to only one feature in the view. Two views are compatible if and only if both views are correct with respect to
the visual model. Depending on the type of output (ADL, behavior protocol, or real-time model) and defined formal transformation rules, the tool will analyze the transformation process and produce the result to the user.

Currently, the transformation analysis is implemented by reversing the transformation process and validating the resulting output with the original XML file that specifies the system. This is done by defining an XSLT with reverse transformation rules, from extended timed automata to system specification in XML. The process takes a behavioral specification as input and produces a system specification as output. Then, the resulted XML file is validated against the original XML file. In this process, every component specification - including its services, attributes, reactivity, data constraints, time constraints, and security specification - should match a component specification in the original XML file. There is no component model that we are aware of which provides model transformation analysis.

4- Simulation and model checking

In [4] we have defined formal transformation rules that transform component types to UPPAAL extended timed automata. We have successfully implemented and tested it on “Boiler Plant” example [17], a benchmark case study in real-time reactive systems area. The distinct advantage of the compiler tool is that it can generate the behavior model in different notations, thus allowing different model checkers to be integrated into the framework to perform formal verification. There exists no general purpose model checker, in the sense that no existing model checker has the ability to model check any stated property. Since trustworthiness attributes can be defined differently by developers for different applications, a developer must be given the facility to plug in the model checker that is most suitable for verifying the chosen trustworthiness properties. This is the rationale behind our design decision to translate the model into different formal notations.

By design the translator in our tool is syntax-directed and hence extensional. The translator in our compiler will only require the grammar of the target language to produce an output in the target language. No change to the translator code is necessary. The major difference between the simulation and model checking process that we have implemented and those of SOFA, Fractal, and SaveCCM is that ours is a unified approach to evaluate timeliness, safety and security.
5- **Real-time analysis**: This tool supports real-time scheduling and real-time analysis relative to criticality, priority, and other real-time non-functional properties. Similar to SaveCCM, we are currently using Times tool [5] to perform real-time analysis. In addition, we are studying the impacts of security considerations on real-time schedule analysis.

6- **Architectural analysis**: This tool analyzes the correctness of the architectural style and system configuration specification relative to architectural constraints defined in the system design.

### 5.2 Implementation tools

7- **Component repository**: This is a storage place to store and reuse developed trustworthy components. The repository provides storage facilities for: (i) component specification (structure and contract), (ii) development source code, (iii) compiled, execution ready assembly of the component, and (iv) usage profiles and certificates. The repository allows storing and retrieving different versions of the same component. SaveCCM, SOFA, and Koala use component repository for component specification and implementation. In comparison, we will use the repository for reusing not only design elements and code but also analysis reports and certificates resulted from the analysis tools.

8- **Code generation**: This tool produces source code. It supports different programming languages such as C++, C#, and java. It analyzes the system design specification. Then, for every component or connector, if the component exists in the component repository then it should reuse it; otherwise, it should produce source code or skeleton for new components. The tool will also develop code for new components by refining existing implementations. The tool provides facilities to use language specific compilers such as C# or java to perform syntactic and semantic analysis of components code. Contracts will be handled as *cross-cutting concerns* implemented as *aspects*.

9- **Traceability and certification**: These tools analyze newly developed components and verify their conformance to design specifications. The traceability tool verifies
that the code satisfies CBD, real-time, and trustworthiness specifications. Traceability analysis is not a trivial task. It requires scrutinizing the generated and developed components code. There are three important types of specifications that should be traced: CBD, real-time, and trustworthiness. We propose the following techniques to perform this operation:

- **Traceability of CBD specifications:** During the automatic code generation or manual development of components, there is a need to maintain the relation between each CBD structural design element and its implementation construct. These relations can be kept in a transformation file in which the name and type of each design element is associated with the name and type of its actual implementation. For example, service names are associated with method names. Then, model transformation analysis techniques can be used to verify the completeness of the code generation and development. At the same time, information can be added to component’s meta data to link the implementation to its source design time specification. For example, current programming languages like C# and java support defining custom attributes. These attributes can add semantic information to implementation constructs such as methods and classes. Then, Reflection techniques are used to read attributes and analyze component’s meta data. Therefore, the traceability tool uses attributes and reflection to analyze the conformance of component’s implementation to its design specifications.

- **Traceability of real-time specification:** Worst case execution time (WCET) of services can be specified as an attribute to a service at design time and as a custom attribute at implementation time. Then, during traceability analysis, the functions that implement real-time services can be executed to check if their measured execution time is bound by their WCET.

- **Traceability of trustworthiness:** The actual traceability of security and safety behavior can be analyzed using run-time analysis techniques described in Section 5.3.

After traceability analysis, the tool interacts with a certification authority to obtain a certificate that indicates the trustworthiness of the component and the level of development conformity to design and quality attributes stated in its specifications. The certificate is issued based on: (i) design-time analysis using verification
techniques of component’s design relative to its specification, and (ii) traceability analysis of component’s implementation relative to its design. After certification, the tool stores certified components along with their analysis reports in the component repository. The certification authority receives: (i) the information of the software development firm, (ii) component’s design specification, code, and analysis reports, and (iii) detailed information about the tools used to generate the analysis reports. Then, the certification authority can verify the claimed analysis reports (may perform the verification and traceability checks again) and issue the certificate. The traceability analysis and certification are distinctive features of our approach.

5.3 Run-time tools

10- Run-time environment: This tool supports running systems and dynamically reconfiguring executions. The tool is a middleware between the component repository and the run-time environment that communicates with the operating system (e.g., J2EE or .NET run-time environment). It communicates with the component repository to load component assemblies. The tool allows a controlled reconfiguration to the running system (e.g., adding a new component or replacing an existing one).

11- Run-time analysis: This tool performs run time analysis during system execution. The tool ensures that system behavior conforms to the stated functional and nonfunctional properties. This is done by observing input, output, and system states during program execution. Execution sequences can be monitored, logged, and visualized to ease analyzing system behavior. These sequences are used to build usage profiles for components. These profiles can be used to monitor the availability and analyze the reliability of components and system. The execution profiles can be subjected to formal verification. Verification is done by ensuring that system execution doesn’t reach a state that violates trustworthiness. It can produce a counter-example in case of system failure.
6 Conclusion

In this paper we have evaluated the current state of the arts of CBSE approaches. The analysis shows that CBSE didn’t reach its objectives and far from fulfilling its promises. Hence, it is unlikely that current practices of CBSE can lead to developing trustworthy systems. Therefore, we have introduced our perspective to remedy the shortcomings of CBSE by proposing a trustworthy component model, a rigorous development process model, and a framework that implements the development process.

As of now, the visual modeling, the compiler, the automatic translator to ADL notation, and translating the model to UPPAAL language for model checking have been completed. We tested the translation to UPPAAL model checker on Boiler Plant case study and verified timeliness, safety and security properties [17]. Times tool has been used to perform real-time analysis. We are optimistic in realizing the rest of the tools.

Component modeling techniques with whom we have compared our work, do not provide all the tools necessary for rigorous analysis at different stages of system lifecycle. The reason is that these component models are designed and implemented for different specific domains. For examples, SaveCCM, Pin, and PECOS are real-time component models. Hence they provide tools for real-time analysis and verification of safety and liveness. On the other hand, SOFA is a distributed component-based model focusing on architectural and communication aspects.

A virtue of the presented framework, when it becomes operational, is that it can be a unified platform for developing component models, regardless of their application domain. It provides both real-time elements and essential architectural features for hierarchical, as well as distributed systems. It is reasonable not to claim that systems developed under this proposed framework will be absolutely trustworthy, but it is justifiable to claim that such systems can be provable to meet the trustworthiness criteria, provided that the tools in the framework are correct.

References


