Toward trustworthy software process models: an exploratory study on transformable process modeling

He Zhang1,∗,† Barbara Kitchenham2 and Ross Jeffery1

1National ICT Australia, University of New South Wales, Australia
2Keele University, U.K.

SUMMARY

Software process modeling and simulation have become effective tools for support of software process management and improvement over the past two decades. They have recently been integrated into the Trustworthy Process Management Framework (TPMF) as the infrastructural components to facilitate the delivery of trustworthy software products. This paper proposes the concept of Trustworthy Software Process Models as inputs to TPMF and introduces transformable process modeling for supporting effective and productive development of trustworthy process models. Furthermore, this paper undertakes an exploratory study on process model transformation by investigating and comparing process modeling semantics between quantitative (e.g., System Dynamics, SD) and qualitative forms of modeling and simulation. By following the model transformation scheme, a quantitative continuous (SD) software evolution process model is successfully transformed into its qualitative form for simulation. The results present the different capabilities and performance between these two modeling paradigms, as well as the possible benefits and interesting perspectives of transformable process modeling.

Received 5 October 2009; Revised 30 May 2010; Accepted 23 June 2010

KEY WORDS: trustworthy process models; transformable process modeling; software process modeling and simulation; qualitative modeling and simulation; system dynamics; software evolution

1. INTRODUCTION

Our daily life has become exceptionally dependent on information technologies and systems. Along with a rapidly increasing number of complex and network-centric software systems, the successful development and deployment of such systems depend more and more on the extent to which we can justifiably trust them. In 2005, the U.S. Center for National Software Studies (CNSS) issued a report, Software 2015 [1], which identified software trustworthiness as the most important focus of future software research.

Software development and evolution are process-intensive undertakings. The delivery of trustworthy software is made through the enactment of trustworthy software processes [2]. In ICSP‡ 2009, a Trustworthy Process Management Framework (TPMF) was proposed as a means for assuring trustworthy software development, in which process models comprise its infrastructural components.

TPMF assumes that software process models are success-critical artifacts for software process management and improvements. The trustworthiness of the software process relates to the extent to which the corresponding process model is trustable. In this paper, we first propose the concept of
trustworthy process model, which conforms to the needs of the adoption of TPMF. Moreover, we suggest that transformable process modeling, as a promising process technology, is an important vehicle to achieve trustworthy software process models.

This paper presents an exploratory study transforming a software evolution process model between quantitative and qualitative modeling paradigms for simulation. Based on our previous exploration on qualitative and semi-quantitative software process modeling and simulation [3–5], an initial model transformation scheme is developed for bridging the gap between quantitative and qualitative software process modeling. The outcome of this study confirms the transformability of software process models and reveals some potential benefits of process model transformation.

This paper proceeds as follows. Section 2 reviews the definitions of trustworthy software and process, and proposes the concept of trustworthy software process model. Section 3 introduces transformable process modeling in support of the development of trustworthy process model. Section 4 starts the exploratory study with developing a transformation scheme between qualitative and quantitative process models. Sections 5 and 6 instantiate the model transformation for simulation using a software evolution process. Section 7 discusses how this approach relates to other research. Section 8 presents our conclusions and suggestions for the future work.

2. TRUSTWORTHY SOFTWARE PROCESS MODELS

2.1. Trustworthy software and process

According to Merriam–Webster Dictionary, trustworthy means ‘worthy of confidence’. For software products, researchers and practitioners have a slightly different understanding of ‘trustworthy software systems’ over time.

An early standard for secure systems (Trusted Computer System Evaluation Criteria, TCSEC) restricted the notion of trustworthiness to dealing solely with security [6]. Trusted Software Methodology (TSM) was developed to provide 44 trust principles and guidance in evaluating trustworthiness during the development process [7].

Since 2000, researchers and practitioners have gained greater understanding of software trustworthiness. Microsoft and Sun Microsystems claimed to have undertaken a Trustworthy Computing initiative. Bill Gates sent a memo to his entire workforce demanding, ‘...company wide emphasis on developing high-quality code that is available, reliable and secure, even if it comes at the expense of adding new features’ [8].

In 2004, over 80 senior executives and thought leaders convened at the second National Software Summit (NSS2) to assess the advance of software. They created a strategic vision ‘achieving the ability to routinely develop and deploy trustworthy software products and systems, while ensuring the continued competitiveness of the U.S. software industry’ [1].

TrustSoft\(^\text{§}\) depicted a holistic view of software trustworthiness with a number of attributes, and developed a multidimensional approach toward trustworthy software systems based on component technology [9, 10]. In 2006, a panel session during COMPSAC\(^\text{¶}\) was held on the theme of ‘Trustworthy Computing’ [11], which aimed to capture all the aspects of trustworthy software and address the needs in the real context.

In ICSP 2009, ISCAS\(^\text{∥}\) and other collaborative research units defined a trustworthy product as ‘a product (software) that satisfies a range of its trustworthiness objectives based on its requirements’, and they also suggested that software trustworthiness is highly dependent on a number of attributes [2].

Table I summarizes the attributes of trustworthy software systems and shows the changes over the past decades. It confirms that trustworthiness is a holistic property. This means the methods for building trustworthy systems must satisfy multiple properties [12], such as the attributes listed

\(^\text{§}\)TrustSoft Graduate School in University of Oldenburg.
\(^\text{¶}\)International Computer Software and Applications Conference.
\(^\text{∥}\)Institute of Software Chinese Academy of Sciences.
TOWARD TRUSTWORTHY SOFTWARE PROCESS MODELS

Table I. Attributes of trustworthy software systems.

<table>
<thead>
<tr>
<th>Source</th>
<th>Trustworthiness attributes</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCSEC</td>
<td>Security</td>
<td>1985</td>
</tr>
<tr>
<td>TSM</td>
<td>Security, reliability, availability</td>
<td>1994</td>
</tr>
<tr>
<td>Microsoft</td>
<td>Available, reliable, secure</td>
<td>2002</td>
</tr>
<tr>
<td>NSS2</td>
<td>Security, safety, reliability, survivability</td>
<td>2005</td>
</tr>
<tr>
<td>TrustSoft</td>
<td>Correctness, security, availability, reliability, performance, security, privacy</td>
<td>2005</td>
</tr>
<tr>
<td>COMPSAC</td>
<td>Availability, reliability, security, survivability, recoverability, confidentiality, integrity</td>
<td>2006</td>
</tr>
<tr>
<td>ICSP</td>
<td>Functionality, reliability, safety, usability, security, portability, maintainability</td>
<td>2009</td>
</tr>
</tbody>
</table>

In Table I. Based on the cumulative understanding of software trustworthiness and the advance of software process technology, a **trustworthy process** is defined as ‘a process that is capable of producing a range of trustworthy products’ [2], which is employed as the default definition in this paper.

### 2.2. Trustworthy process management framework

In [2], **process trustworthiness** is defined as ‘a capability indicator to measure the relative degree of confidence for certain software process to deliver a trustworthy product’. A TPMF was proposed in order to facilitate the management and assurance for a trustworthy development process.

The core of TPMF comprises the Software Trustworthiness Model, Process Trustworthiness Model, and Measurement Model, which are supported by four infrastructural components: Requirement Management Model, Process Management Model, Risk Management Model, and Process Simulation Model [2]. On the basis of this framework, the successful delivery of trustworthy software product is dependent to a large extent on the ‘trustworthiness’ of these component models.

### 2.3. Trustworthy software process model

Osterweil takes the view that ‘software processes are software too’ in [13, 14]. From this perspective, software processes are a subset of software. Thus, like design documents and source code, process models, viewed as abstractions of the real-world software processes for specific objectives, are a type of software artifact. With reference to the dependencies specified by TPMF [2], their trustworthiness is a critical factor leading to the delivery of trustworthy software systems.

From this perspective, it is not difficult to reformulate the concept of **trustworthy process models** based on the definitions introduced in Section 2.1. A trustworthy software process model is a model capable of representing a realistic software development or maintenance process at a certain level of abstraction that satisfies its modeling objectives and requirements. Note that a trustworthy process model does not imply a trustworthy software process, e.g., a trustworthy process model may predict project failure in terms of the inputs given. However, we believe that the implementation of trustworthy processes does require the support of trustworthy process models.

By carefully reviewing the appropriateness of the attributes summarized in Table I, we suggest that the trustworthiness of process models is related to the following set of properties which have been redefined to address this context:

- **Safety**—inclusion of semantics that represent process requirements related to safety, and the ability to highlight inconsistencies in the process model with respect to safety-related processes.
- **Validity**—the ability of the process model to reflect the assumptions and constraints about the software process specified by process stakeholders.
- **Reliability**—the probability of the process model delivering results that is consistent with the model assumptions.
- **Reusability**—the ability to obtain or build the process model for the given software process and objectives from the related work.
• **Scalability**—the ability of the model to fully represent all the required processes, complex or large scale, at the required level of granularity.
• **Maintainability**—the ease with which the process model can be adapted to the changes in the software process being modeled.
• **Performance**—effectiveness of the model construction that is able to give a quick response or reaction with minimal resources and/or time taken.

3. TRANSFORMABLE PROCESS MODELING

Once the concept of trustworthy software process model is proposed, we need to know how to achieve it. This section proposes *transformable process modeling* driven by *trustworthy software process models*.

3.1. Multi-dimensional transformability

Currently, processes and modeling paradigms (languages) are monolithic and heterogeneous. Our systematic review on software process simulation modeling identified 10 process modeling paradigms for simulation study [15]. Furthermore, a recent boarder-scoped systematic review reported more than 30 methods employed for software process modeling and analysis [16].

Correspondingly, the diversity of software processes and process models is even larger. Depending on the modeling purposes and approaches, the process can be viewed at varying abstraction levels in different dimensions. For the objective of research, software process models may be considered as *static* or *dynamic*, *qualitative* or *quantitative*, *black-box* or *white-box*, *prescriptive* or *descriptive* [17].

Instead of inventing new process modeling paradigms or languages, transformable process modeling focuses on *multi-dimensional transformability* of software process models and researches the conditions that enable such transformations. Figure 1 illustrates the modeling diversity by depicting process models in a 3-dimensional transformation space as an example. Each cell indicates one instance of a process model for the specific software process. A process model is able to become transformable in this space if the semantic relationships and completing conditions among the modeling paradigms can be explicitly identified.

For instance, Osterweil identified two levels of process research: *macro-process* research, and *micro-process* research [18], which are related to the *granularity* dimension in transformation space (Figure 1). The former may be instantiated as cause-effect and analytic modeling from project to organization level, providing indicators for high-level decision making; whereas, the latter is more...
consistent with the prescriptive and descriptive process definition, tailoring the execution from finer-grained activities to project level.

On the semantics dimension, a process can be represented by different modeling languages on an identical granularity level. In micro-process research, for example, a process is able to be described by a number of discrete modeling languages, such as Little-JIL, SPEM**, which provide the comparable modeling capability on a fine-grained abstract level, but vary in their featuring semantic representation and richness.

Software processes, on the precision dimension, can be modeled on different certainty levels as well. Note that the precision dimension does not imply accuracy. For instance, some modeling paradigms may produce precise process models and outputs, but their accuracy has to be further assessed. The exploratory study described in the remainder of this paper provides an example of process model transformation on the precision dimension.

Apart from these three major transformation dimensions, there might be other dimensions as well, such as discrete or continuous (i.e., approach to time) and prescriptive or descriptive.

3.2. Process model transformation

The process model instances residing in the transformation space are constrained by multiple dimensions. Crossing process scopes, software process models representing the same or similar process may become transformable depending on their goal level (granularity), semantic richness and information precision (as shown in Figure 1).

For instance, in the transformation space, research may answer questions such as ‘how does a representation of a process in one notation (form) relate to the representation in another notation?’, ‘what conditions or constraints are needed for such transformation?’, or ‘how do we link them to reflect the concerns at different levels in an organization?’

As shown in Figure 2, transformable process modeling analyzes the semantic equivalence at the following three levels, where the model transformation may take place.

Elementary semantics. A process model is constructed on the basis of a set of atomic elements that represent the semantics of the modeling paradigm or language employed, such as level and flow (rate) in System Dynamics (SD) or place and transition in Petri-net. The transformation of process model starts with establishing the correct semantic mapping between two modeling paradigms on the element level.

Structural semantics. Although a process model is built on the basis of modeling elements, it is not sufficient to simply assemble components that are themselves trustworthy [19]. Structural semantics seek to define the mechanism by which modeling elements are built upon each other to form a valid process model, and try to replicate the equivalent mechanism with other modeling techniques or languages.

---

**Software Process Engineering Meta-model.
Composable semantics. Between elementary and structural semantics, there are process fragments of intermediate scale composed of several elementary elements. Such components are patterns that can be reused in many different process models. They can themselves be assembled to represent a larger process.

For example, Madachy identified two levels of reusable patterns of SD process model [20]: generic flows which are the smallest, essential structures based on a modeling elements (rate/level) that model the common situations and characteristics (e.g., draining process and production process); and infrastructures and behaviors which are based on one or more of the generic flow types with additional structural details (e.g., delays and oscillation). In [21], Garousi et al. also identified process macro-patterns that can be reused in modeling and simulating different phases of V-model-type software development.

Note that transforming a process model between modeling paradigms does not mean unconditional or reversible (symmetrical) transformation. It depends on the semantic richness and equivalence level between two paradigms. In many cases, the completing information is required for some directions of transformation, such as from qualitative to quantitative. However, transformable process modeling is able to maximize the reuse of knowledge at the above three semantic levels between process models. The following exploratory study addresses the related issues in more detail.

3.3. Model transformation supporting trustworthy process models

Trustworthy process models, like trustworthy software products, must satisfy a variety of properties (Section 2.3). Unfortunately, each of these properties has its own modeling emphasis, which makes it hard to reason about a process model that needs to satisfy multiple properties.

The goals of transformable process modeling for trustworthy software process models are as follows:

- To accelerate the trustworthy process (model) development in support of trustworthy process management.
- To maximize the reuse of trustworthy software process models in multiple scales and across the barriers between modeling techniques.
- To reduce the incidental modeling errors related to developing process models from scratch.
- To minimize the effort of trustworthy process model development and maintenance by relieving the burden of learning other transformable modeling techniques.

Figure 3 illustrates the properties of trustworthy process models that transformable process modeling is able to support, and illustrates the supporting relationships between them.

Validity. Transformability may enable cross-validation to reconstruct the assumptions and constraints from another transformed version of the model and check that all assumptions and constraints are explicitly equivalent to those specified by the stakeholder. In addition, by implementing validated model semantics or even whole process models transformed from other paradigms, the effort for validating new process models and/or semantics can be minimized.

Reliability. A process model’s semantics may be tested within different organizational contexts or process settings. If process models become transformable between modeling paradigms, process engineers and researchers would be able to use their preferred modeling paradigm increasing the productivity and the model quality.

Reusability. The ability to transform process models into other formats will increase the extent to which process models and process fragments can be reused. With the support of process model transformation schemes, many existing process models (or components) will become instantly available for research and practice in other technical or organizational contexts.

Scalability. Model transformation will increase the interoperability between process models (or components) devised with different modeling paradigms. This allows an easy integration between models at different levels of granularity and different precisions.
Maintainability. Transformable process modeling enables process engineers to maintain the process models and components that were developed with the modeling techniques they are not familiar with.

4. TRANSFORMING QUALITATIVE AND QUANTITATIVE PROCESS MODELS

The exploratory study was initiated by our previous research on qualitative and semi-quantitative modeling of software processes [3, 4, 17]. In this paper, we develop a model transformation scheme between qualitative and quantitative (continuous) software process models, i.e., a transformation in the dimension of precision (Figure 1).

4.1. Quantitative and qualitative process modeling

This subsection briefly introduces the selected qualitative and quantitative process modeling paradigms for transformation in this study.

4.1.1. Qualitative process modeling. Qualitative models reflect the systems in the real world at an abstract level. Fewer assumptions are required than for quantitative models. Qualitative simulation is implemented in the QSIM tool [22].

A system can be modeled quantitatively as a set of ordinary differential equations (ODEs). At a coarser level of abstraction, a qualitative differential equation (QDE) represents a large set of possible ODEs, for example, each $M^+$ (or $M^-$) function represents the set of all monotonically increasing (or decreasing) functions. When only incomplete knowledge is available, we can replace ODEs with QDEs to represent the relationships and values of the variables qualitatively [23].

Qualitative modeling is able to cope with a lack of precise knowledge by modeling processes at a more abstract level than quantitative modeling. The output generated by QSIM is a set of possible qualitative behaviors and each behavior consists of a sequence of states. Time is treated as a qualitative variable in QSIM. Semi-quantitative simulation is a quantifiable extension of qualitative simulation.

4.1.2. Quantitative process modeling paradigm. Most software process models for simulation are purely quantitative, and require detailed understanding and accurate measurement of software processes. Software processes can be modeled quantitatively using various techniques, e.g.,
continuous or discrete modeling paradigms. The selection of quantitative modeling paradigms for transformation and comparison is the first and most important step in this exploratory study. According to the results of our systematic review on Software Process Simulation Modeling (SPSM) [15, 24], SD and Discrete-event Simulation (DES) are the most widely used quantitative continuous and discrete modeling techniques. Furthermore, qualitative modeling approaches can also be regarded as a type of continuous modeling based on differential equations. For this exploratory transformation, therefore, SD is selected as an appropriate quantitative modeling approach to be the counterpart of qualitative modeling in this study.

The remainder of this section develops the model transformation scheme between quantitative (SD) and qualitative paradigms at structural, elementary, composable levels.

4.2. Structural semantics: causal loop diagram

In SD, several modeling components and tools are used to capture the structure of systems, including the Causal Loop Diagram (CLD), level and rate, and delay. Among them, CLD (also referred as influence diagram) is well suited to represent the structural semantics (interdependencies and feedbacks) in the continuous process model. A CLD consists of variables connected by arrows denoting the causal influences among the variables. Each causal link (arrow) might be assigned a polarity, either positive (+) or negative (−) to indicate how a dependent variable changes when the independent variable changes. The causal links are quantified in SD, but the CLD does not reflect such quantification.

In qualitative diagramming, i.e., Abstract Structure Diagram (ASD) in QSIM [22], the notations and links are more explicit and clear. The sum and product relations are explicitly represented by add (or sum) and mult identifiers. Other basic arithmetic relations, e.g., subtraction and division, can also be derived from them. Complicated or unknown positive (+) and negative (−) dependencies can be denoted as the monotonic increasing $M^+$ and decreasing $M^−$ functions in qualitative modeling [22]. Therefore, a CLD can be transformed into its corresponding qualitative diagram, but is incomplete without additional information.

4.3. Elementary semantics: level and rate

Level (or stock) and rate (or flow) are the central concepts of SD modeling semantics. Levels absorb inflow rate, and accumulate the difference between the inflow to a process and its outflow. SD uses a particular diagraming notation for levels and rates (Figure 4(a)). Valves control the flow rates. Clouds represent the sources and sinks for the flows, which are both outside the model boundary.

The elementary semantics represented in Figure 4(a) correspond exactly to the following integral equation:

$$\text{level}(t) = \int_{t_0}^{t} [\text{inflow}(s) - \text{outflow}(s)] \, ds + \text{level}(t_0)$$

(1)
Equivalently, the net rate of level change, its derivative, is the inflow less the outflow, defining the differential equation:

$$\frac{d}{dt} \text{(level)} = \text{inflow}(t) - \text{outflow}(t)$$

(2)

Hence the SD model in Figure 4(a) can be modeled in QSIM by a differential equation [22] with QDE semantics. Unlike SD diagraming, the rate difference (net flow) has to be explicitly represented in qualitative diagram. Figure 4(b) shows the equivalent QDE (corresponding to level and rate) in a qualitative modeling diagram.

4.4. Composable semantics: delay

When modeling software processes in SD, a number of composable components can be reused within or between process models [20, 21]. Among them, Delay is a semantic component different from other generic flows and infrastructures and behaviors [20], which is frequently used in SD modeling. Forrester identified two characteristics of a delay [25]. One is the length of time expressing the average delay \( D \), which fully determines the ‘steady-state’ effect of the delay. In the steady state, the flow rate multiplied by the average delay gives the quantity in transit within the delay. The other is the ‘transient response’, which defines the functional relationship between time and outflow. Delays with the same average delay time \( D \) can have quite different transient responses to changes in input rate (see plots (a) and (b) in Figure 5).

Exponential delay is the most frequently used delay in SD. Figure 5 shows two common types of exponential delay used in software process modeling: first-order delay (a) and third-order delay (b). Mathematically speaking, an \( n \)th order delay is equivalent to \( n \) cascaded single-order delays, with each having a delay time of \( \frac{D}{n} \) [25].

As time is treated qualitatively in qualitative simulation, there are only ‘critical time points’ as landmarks of a modeled process, but no numeric time periods between them. The QSIM algorithm [26] does not explicitly consider delay phenomenon, nor does it include a built-in delay function. However, as Semi-quantitative Simulation (SQSIM) offers the capability of handling numeric values, there is no reason why a delay should not be included in an SQSIM model. Unfortunately, as far as we know, there is no such semantic element built in any available qualitative simulation packages. This subsection demonstrates how to implement (exponential) ‘delay’ semantics in the QSIM framework.

4.4.1. First-order delay

First-order and third-order exponential delays are two of the most common delays used in the SD models of software process. Figure 6 is a first-order delay presented in SD diagraming. Given an exogenous inflow rate (IN from another part of system), a first-order delay consists of a simple level (LEV) and a rate of outflow (OUT) that depends on the level and on the delay time (DEL). Table II shows the mathematical equations of first-order delay.

A first-order delay is composed of four model elements (Figure 6): one level, two rates, and one auxiliary variable (delay). Following the elementary transformation scheme, the structure of first-order delay is converted and represented with ASD notations (Figure 7(a)). A new ASD notation (Figure 7(b)), with two inputs (inflow and delay) and one output (outflow), is created to abstract this structure and avoid the redundant complexity of the qualitative model.
### Table II. First-order exponential delay.

<table>
<thead>
<tr>
<th>OUT ([i,i+1])</th>
<th>(=\text{DELAY1}(\text{IN} [i,i+1], \text{DEL}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT ([i,i+1])</td>
<td>(=\text{LEV}[i]/\text{DEL})</td>
</tr>
<tr>
<td>(\text{LEV}[i])</td>
<td>(=\text{LEV}[i-1]+(\text{DT})(\text{IN}[i-1,i]-\text{OUT}[i-1,i]))</td>
</tr>
</tbody>
</table>

| \(\text{OUT}[i,i+1]\) | The outflow rate between time \(i\) and \(i+1\) |
| \(\text{LEV}[i]\) | The level stored for delay at time \(i\) |
| \(\text{DEL}\) | The average delay time |
| \(\text{DT}\) | The time step between successive evaluations of equation |
| \(\text{IN}\) | The inflow rate between time \(i-1\) and \(i\) |

### Table III. Third-order exponential delay in SD.

<table>
<thead>
<tr>
<th>OUT ([i,i+1])</th>
<th>(=\text{DELAY3}(\text{IN} [i,i+1], \text{DEL}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{R1}[i,i+1])</td>
<td>(=\text{LEV1}[i]/(\text{DEL}))</td>
</tr>
<tr>
<td>(\text{LEV1}[i])</td>
<td>(=\text{LEV1}[i-1]+(\text{DT})(\text{IN}[i-1,i]-\text{R1}[i-1,i]))</td>
</tr>
<tr>
<td>(\text{R2}[i,i+1])</td>
<td>(=\text{LEV2}[i]/(\text{DEL}))</td>
</tr>
<tr>
<td>(\text{LEV2}[i])</td>
<td>(=\text{LEV2}[i-1]+(\text{DT})(\text{R1}[i-1,i]-\text{R2}[i-1,i]))</td>
</tr>
<tr>
<td>(\text{OUT}[i,i+1])</td>
<td>(=\text{LEV3}[i]/(\text{DEL}))</td>
</tr>
<tr>
<td>(\text{LEV3}[i])</td>
<td>(=\text{LEV3}[i-1]+(\text{DT})(\text{R2}[i-1,i]-\text{OUT}[i-1,i]))</td>
</tr>
</tbody>
</table>

\(\text{LEV}[i]\) | \(=\text{LEV1}[i]+\text{LEV2}[i]+\text{LEV3}[i]\)

#### 4.4.2. Third-order delay.

A third-order delay is the equivalent of three first-order delays cascaded one after another, so that the output of the first is the input to the second, and the output of the second is the input to the third. Figure 8 illustrates the structure of a third-order delay in SD diagraming. Table III shows the equations for calculating a third-order delay.

By applying the new ASD notation created for first-order delay (Figure 7(b)), the structure of a third-order delay can be represented in Figure 9(a). Again, another new ASD notation (Figure 9(b)), with two inputs and one output, is created to abstract this more complicated structure. Here, the
Figure 9. Implementation of third-order delay in ASD: (a) implementation and (b) notation.

Implemented third-order delay also demonstrates how to construct an \( n \)th-order delay using the basic first-order delay in QDE (based on QSIM algorithm framework).

5. REFERENCE PROCESS MODEL FOR TRANSFORMATION

This section introduces the reference model used in Section 6 to illustrate the process of transforming from a systems dynamic representation of a model into a qualitative or semi-qualitative version.

5.1. Selection of reference process model

As software evolution was identified as one of the most interesting topics in SPSM research by our systematic review [15, 24], this paper instantiates the model transformation with a reference model of a software evolution process.

The reference model needs to contain the most common elementary, structural and composable semantics of the chosen evolution process. Nevertheless, the overall model’s structure should be clear and simple enough to ensure that the emphasis of this research is on model transformation, rather than construction of a complicated model.

According to the above criteria of the quantitative reference model, an SD model of software evolution process was selected for this study. There are several candidate models published in the past decade [27–31]. Among them, Wernick and Hall’s model [31] consists of a single module, and is the most recent SD model of evolution process found in our SLR.

5.2. Software evolution process

The insights obtained from the previous studies indicated that software evolution could be systematically studied and exploited using SPSM approaches. They also suggested that to some extent software evolution is a disciplined phenomenon as illustrated, for example, by the regularity of functional growth patterns [32]. Models of such patterns permit the forecasting of future overall system growth and growth rates. Moreover, the observed patterns of behavior yielded common phenomenological interpretations. Basically, four important feedback structures, identified by the previous related work, are used in the model construction of software evolution processes:

- **Inertia-like (anti-regressive) effect due to system growth.** The first hypothesis is that increasing the size of a software system and changes in unanticipated directions will over time reduce the enhanceability and modifiability of that system [30]. These changes may result in a decay in software architecture. Meanwhile, new changes also have to be incorporated into an existing system structure, and as the software grows, there are more existing components into which each new change needs to be fitted. As a result, software developers are required to spend time on tasks specifically intended to maintain the system structure, and to compensate for the software aging effects, which are referred to as ‘anti-regressive’ activities [29].

- **Effects of decreasing knowledge coverage.** The increasing complexity of a software system also reduces the developer’s ability to change the system because of a decrease in coverage of the developer’s knowledge of the system components, their composition and interactions [30]. As the software grows, the amount of knowledge needed to support future changes grows as well, but at a faster rate, as the implementation of each new component needs to be seen in the context of all
the existing systems [33, 34]. If the developer’s knowledge does not grow at this rate, it may fall behind the knowledge needed to support further changes.

**Generation (progressive effect) of new requirements.** The release of upgraded software with new functionalities enables users to exploit opportunities for novel or extended system use, which in turn result in demand for further functionalities [28]. This positive feedback is recognized as ‘progressive’ type of work, which enhances software functionality by modification of or addition to the code and/or the documentation [29].

**Correction of implementation faults.** After software is released and deployed, some requirements may be found to be incorrectly implemented. They are eliminated from the specification, but may be replaced in the requirements with new or changed equivalents [27]. These need to be corrected. However, it is possible for the implemented corrections themselves to be wrong. Thus, the process of correcting requirements is represented by a rate of successful corrections and a rate of unsuccessful corrections.

5.3. **A simplified quantitative model**

**5.3.1. Model description and calibration.** The reference quantitative model (shown in Figure 11) was developed using Vensim simulation environment (Ventana Systems, Inc.). Although it is a simplified model, it incorporates three of the typical feedback loops described above (indicated by the numbers). They are feedbacks representing the inertia effect (anti-regressive activities, Loop 1), the generation of new requirements (progressive activities, Loop 2) and the correction of faults in previous implementations (Loop 3).

The ‘size’ of software system has been abstracted into a number of arbitrary-sized ‘units’ of requirements, since it is a more informative reflection of software evolution, which is more likely driven by changes in functionality than by low-level ‘code’ considerations [33]. Plus, it avoids issues related to a specific size metric.

The delay used in this model is a third-order delay, which fits the technical software process [27]. It represents the time delays caused by some entity passing through the phases of a process made up of a sequence of sub-processes, each of which depends for its input on the output of the previous one.

The calibration inputs to the reference model are based on actual data for the evolution of the ICL VME mainframe system described in [28].

**5.3.2. Sensitivity to policy change.** The reference model can be subjected to a sensitivity analysis to investigate the effects of changes in policy inputs. Wernick and Hall introduced five policy factors to the reference model, each of which varies from its default value of 1, using a normal distribution with a standard deviation of 0.25. For instance, values greater than 1 of ‘inertia scaling policy’ indicate higher maintainability and evolvability of the system.

To simplify the discussion, three of their policy factors (i.e., inertia scaling policy, new requirement scaling policy, and fault generation scaling policy) underlined in Figure 11) are selected to investigate the policy sensitivity of three feedback loops respectively. Figure 12 shows the distributions of simulated system size (SImp, requirements implemented in Figure 11) growth and volume of requirements (SReq, requirements to implement in Figure 11) over time (using the replicated model) for 1000 runs for each parameter varied separately. The solid line in each plot indicates the mean result, and the regions on either side of it contain 50, 75, 95 and 100% of the simulated results respectively.

6. TRANSFORMED QUALITATIVE AND SEMI-QUANTITATIVE PROCESS MODELS FOR SIMULATION

**6.1. Transformed qualitative and semi-quantitative process model**

As the first step of qualitative modeling, the qualitative assumptions need to be abstracted from the real-world system [22]. However, due to the quantitative (SD) reference model available, an inverse
procedure can be followed. The SD model can be transformed into a qualitative model based on
the scheme developed in Section 4. After that, the qualitative assumptions can be extracted from
the corresponding qualitative model.

Figure 13 shows the corresponding qualitative model. As mentioned earlier, it is not necessary
to model and simulate delay semantics with a ‘qualitative’ length. Thus, the qualitative model does
not explicitly include the delay semantics as in the reference model. Based upon the transformed
qualitative model, the reference model’s inherent qualitative assumptions can be explicated:

(1) Requirements to implement ($S_{Req}$) come from exogenous requirements, new requirements
feedback and incorrect requirements feedback;
(2) $S_{Req}$ is transferred to Requirements implemented ($S_{Imp}$) at software developing rate ($R_{SD}$);
(3) The incorrectly implemented requirements, as a small portion of $S_{Imp}$ is returned to $S_{Req}$
for rework;
(4) Increasing existing system size ($S_{Imp}$) incurs more effort needed for ‘anti-regressive activi-
ties’, and decreases $R_{SD}$;
(5) The input effort ($R_{En}$) has linear relationship with $R_{SD}$;
(6) The new requirement feedback ($R_{New}$) has a linear relationship with $R_{SD}$;
(7) The incorrectly implementation ($R_{Inc}$) has a linear relationship with $R_{SD}$;
(8) The development team size does not change (neither recruitment nor turnover) during the
evolution process;
(9) There is no exogenous requirements ($R_{Exo} = 0$) during the evolution process††.

Figure 14 shows the semi-quantitative model based upon the graph of the qualitative model and
the introduction of more semantic constraints (e.g., delays). The newly introduced notations ‘D3’
representing the exponential delays (Section 4.4) are included in semi-quantitative constraints.

One monotonic function ($M$–, between $f_{ie}$ and $S_{imp}$) is included in the qualitative model. This
nonlinear relation needs to be quantified at this stage. The inertia factor was quantified as a multiple
of the inverse square [30] or inverse cube [31] of the existing system size respectively. Therefore,
an envelop function (Equation 3) is constructed for this case.

$$f_{ie} = \left( \frac{\hat{\lambda}_1}{S_{imp}^3} \right) + \left( \frac{\hat{\lambda}_2}{S_{imp}^2} \right)$$ (3)

where $\hat{\lambda}$ is a suitable constant, and determined from historic data.

6.2. Qualitative simulation

The reference model terminated the simulation at a predefined time point (the 156th month). However,
as time is treated qualitatively in the transformed qualitative model, this quantitative
termination condition cannot be implemented in QSIM. Moreover, oscillation phenomena are
observed in some simulated behavior patterns. Thus, the qualitative simulation cannot stop itself
on these behaviors until it runs out of memory. In this study, the simulation was set to halt
after generating a ‘large’ number of behaviors, which is sufficient to observe the major behavior
patterns.

The qualitative simulation generates a diversity of behaviors of the evolution process, most of
which are the combinations of varying patterns of important variables. Figure 15 shows the most
typical behavior patterns of some important variables. Note that the plots generated by QSIM only
depict the behavior trends at the critical time points ($T_0, T_1, \ldots$) when changes happen regardless
of the distance between the time points. The upward and downward arrows indicate the changing
trends of the variables of interest at the critical time point.

Requirements implemented. The simulated behavior of $S_{Imp}$ presents a growing trend at all
times (Figure 15(a)), which is also quantitatively reflected in Figure 10 and Figures 12(a), (c), (e).

††This assumption is directly transferred from the reference model.
H. ZHANG, B. KITCHENHAM AND R. JEFFERY

Figure 10. Simulation of implemented requirements over time: (a) size by the reference model and (b) size by the replicated model.

Figure 11. Quantitative reference model of software evolution.

It is monotonically increasing because developing software is assumed to be always greater than incorrectly implementing.

Requirements to implement. The possible changes in $S_{\text{Req}}$ are more complicated than $S_{\text{Imp}}$. It may gradually drop (down to zero), then rebound to a certain level (Figure 15(b)). Overall, its behaviors generated through qualitative simulation cover all the behavior patterns from the reference SD model (Figures 12(b), (d), (f)). In addition, the qualitative behaviors imply that an upward trend is also possible toward the end of the evolution process (the two rightmost plots in Figure 15(b)).

Requirement implementation rate. It is easy to identify the oscillation of $R_{\text{Imp}}$ from the simulated possible behaviors (Figure 15(c)). The evolution process may reach the equilibrium state after one, two, three oscillations or more, even keep oscillating for ever, which is one of the main reasons that the simulation cannot stop itself. The oscillation phenomenon is to a large degree consistent with the qualitative behaviors observed by Ramil and Smith’s study [35], which constructed a qualitative simulation model based on an analytic model, instead of transformation from a continuous causal model.
Figure 12. Sensitivity of policy change for reference model: $S_{imp}$ by inertia; (b) $S_{req}$ by inertia; (c) $S_{imp}$ by new requirement; (d) $S_{req}$ by new requirement; (e) $S_{imp}$ by fault generation and (f) $S_{req}$ by fault generation.

Figure 13. Corresponding qualitative model of software evolution.

Figure 14. Corresponding semi-quantitative model of software evolution.
Requirement generation rate. Figure 15(d) reflects the ‘unpredictable’ behavior of $R_{Req}$. Qualitatively speaking, for example, the addition of a ‘positive number’ and a ‘negative number’ produces an ‘unpredictable’ result in qualitative simulation, because the sum might be positive, negative or zero. In this case, the variable may oscillate across, over or below zero. (The negative ‘$R_{req}$: net implement rate’ in Figure 15(d) indicates that requirement implementation is faster than requirement generation) Hence, the quantitative constraints have to be applied for $R_{Req}$ (such as the value range constraints later applied in semi-quantitative simulation) to generate more specific and stable behaviors.

Other variables, such as $R_{SD}$, $R_{inc}$, $R_{new}$, and $f_{ie}$, keep decreasing from the start of the simulation. In some cases, they can finally reach an equilibrium state (Figure 15(e)).

6.3. Single-point value simulation

In the following subsections, we compare the quantitative simulation (SD) and the SQSIM from two aspects: simulation with single-point value and simulation with a value range. Traditionally, purely quantitative simulation always assigns a single-point (numeric) value for each input variable during each simulation run [22]. Although sensitivity analysis has been suggested to SD modelers [36], it usually involves investigating the impact of varying the numeric values of certain parameters of interest and performing multiple runs of the simulation. In contrast, semi-quantitative simulation treats a range of values in the same way as single-point values are treated in the quantitative approach. Thus, to implement a single-point value simulation within the semi-quantitative approach, the upper and lower bounds should be set to the same value.

Even with single-point settings, SQSIM generates nine behaviors. Although the simulation is preset to terminate at the 156th month (as within the SD model), the $Q2$ algorithm also includes some behaviors that may terminate in a range covering the preset termination point, e.g., [29.6, 156] months. This is because SQSIM is based on value range calculation and reasoning, rather than single-point value calculation in classical mathematics. All behaviors covering this termination condition are generated by the $Q2$ algorithm. As we are only interested in the behaviors that terminate exactly at the preset time point, we can remove the ‘invalid’ behaviors, and are left with two behaviors that terminate at Month 156.

Figure 16 depicts the trends of some variables of interest in the two behaviors, which directly relate to the ‘size’ changes during evolution. The trends are consistent with the upward trend of $S_{imp}$ and with the downward trend of $S_{Req}$ found in the quantitative simulation (Figure 12) and qualitative simulation (Figure 15). The only difference between them is that $S_{Req}$ may reach zero or remain in the range [0, 50] units when the simulation terminates. The changes to the other variables appear the same (shown in Figure 16). The quantitative simulation in Section 5 predicts that the system size grows up to 433.75 units at the 156th month. Both the valid behaviors from SQSIM produce the close value range, [433, 434] units, for the system size, $S_{imp}$.

Compared to the simulation result of SD (presented in Figure 10), the graphical results from SQSIM only depict a monotonic trend for $S_{imp}$, but the graph lacks a well-defined shape. The shape depends on the number of landmarks created in the course of simulation and can be enriched by inserting more landmarks (like $Q3$ algorithm).

6.4. Value-range simulation

In Section 5, several policy factors were introduced for sensitivity analysis (Figure 11). This subsection compares the value ranges found by the Monte Carlo simulations of these inputs in terms of probability distributions and SQSIM with the corresponding value ranges. The policy factors need to be added in the semi-quantitative model. However, their probability distributions are not necessary in this form of simulation.

The results of SQSIM are summarized in Table IV and compared with the SD results. The value ranges on the second row are the simulated results generated by normal $QSIM + Q2$ algorithm.

\[\text{The quantitative extension to QSIM enables semi-quantitative simulation [22].}\]
Although they are consistent, it is clear that the ranges of SQSIM are very coarse. This is mainly because the evolution behaviors ($S_{imp}$) are monotonic and smooth. They include no transition points and few critical time points either to generate finer ranges or to reduce the uncertainty.

To improve the performance of SQSIM, the Q3 algorithm, which optimizes Q2 with step-size refinement [22], was further applied to obtain finer-grained value ranges. The simulation results (with the step-size of 10) are shown on the bottom row in Table IV. It demonstrates that the
accuracy of SQSIM can be improved significantly by adaptively introducing additional landmarks. The remaining difference between these two approaches is probably caused by (1) the different reasoning mechanism of SQSIM, which is based on the behavior chattering technique instead of single-point calculation used in SD; (2) the sampling and assumed probability distribution (in the quantitative simulation) cause some missing points; (3) the step-size of Q3 has not been completely optimized. Nonetheless, using the Q3 and other advanced refinement techniques (e.g., dynamic envelopes), semi-quantitative simulation can smoothly span the gap from qualitative simulation on the one hand to quantitative simulation on the other [22].

6.5. Summary of transformation between quantitative and qualitative process models

Transformable process modeling enables the transformation and comparison of models constructed with different modeling paradigms. Overall, our results show that both qualitative (semi-quantitative) and quantitative modeling (SD) have their strengths and weaknesses. During modeling, in contrast to causal loop diagraming, the qualitative approach starts with explicitly stated qualitative assumptions, and then transforms them into more specific and clearer constraints. Thus, it provides a more rigorous approach than a CLD. Furthermore, a CLD model does not offer simulation capability, whereas a QSIM model does. Both CLD and QSIM models can be quantified to become their quantitative or semi-quantitative counterparts.

During simulation, QSIM is capable of generating the typical ‘qualitative’ process trends without any quantification. SQSIM refines the process behaviors from QSIM with ‘loose’ quantification. Both SD and SQSIM can produce similar results with single-point values. However, SD presents the variation of a variable in more detail than SQSIM. The SQSIM approach reflects trends more qualitatively, because it uses the same plotting mechanism as QSIM. When dealing with uncertainty, the value range (or discrete values) and their associated probability distribution are often required for any stochastic (quantitative) simulation. However, the sampling mechanism used in quantitative simulation may miss some important values within the given range. In contrast, the semi-quantitative approach handles uncertainty with value range and envelop function, and guarantees the coverage of all possible simulation results.

In brief, qualitative (semi-quantitative) simulation enables simulation studies in a situation where specific process knowledge is incomplete or uncertain. Along with process improvement and knowledge accumulation, a qualitative model can be transformed into a quantitative model for simulation. In the opposite direction, a quantitative simulation model can be downgraded to...
a qualitative (semi-quantitative) model by transformable modeling to suit a new environment or organization. Our previous study discusses the use of qualitative (semi-quantitative) simulation and its integration with quantitative (continuous and discrete) simulation in support of process improvement [37].

Note that, by following the transformation scheme on three semantic levels, transforming a quantitative continuous process model into a qualitative form seems straightforward. In the reverse direction, however, the model variables have to be quantified when transforming qualitative model into a quantitative form.

7. RELATED WORK

This section briefly introduces other emerging research directions in software process modeling that might be related to transformable process modeling, and explains the difference from our approach.

7.1. Software process lines

In [38–40], IESE researchers combined process research and software product line engineering, and developed a conceptual model for Software Process and Product Lines (SPPL), which describes the requirements for defining a variant-rich process in business and manufacturing domains. They also proposed the PESOA (Process Family Engineering in Service-oriented Applications) process that embeds the variant-rich processes for developing, using and maintaining families of processes in both domains.

In [41], Simidchieva et al. added annotated variations to process definitions to enrich precise process details, and proposed a similar set of process variants as a process family. Thus, they introduced a formal approach to defining process families based on characterization of variability.

Although the above software process lines are also able to support trustworthy software process models, all these research programs focus only on the variations between business or manufacturing processes, which are limited to the semantics of a stand-alone modeling paradigm.

7.2. Reusable process models

Madachy proposed model reuse for software process with emphasis on SD [20]. He summarized model structures and behaviors, and organized them in an object-oriented framework. This approach provides a set of process patterns that frequently occur as ‘common assets’ for continuous process modeling, as well as outlines a higher level of abstraction for process modelers. It aims to make process model development easier with reduced effort and fewer errors [42], which conforms to the goals of transformable process modeling addressed in this paper.

Although he addressed reusable process modeling of software process model which is relevant to our approach, his approach limits model reuse to SD only. However, we suggest that reusable process modeling can be integrated into transformable process modeling as an effective methodological component.

7.3. Process patterns

Process patterns are the structural patterns of processes that can be frequently observed and recognized in the process execution. In general, they conform to the composable semantics in transformable process modeling. In [21], Garousi et al. constructed a customizable software process simulator, GENSIM 2.0, based on a limited set of generic process structures (macro-patterns). Our previous research, Process Enactment Analysis [43], enables the recovery of process model from process enactments by mining and identifying execution patterns.

---

1Fraunhofer Institute for Experimental Software Engineering.
7.4. Model conversion

In [44], Mak investigated the possibility of combining DES and SD modeling, and developed a set of rules for converting a DES model into an SD representation.

Although this work was not dedicated to software process modeling, as DES and SD are the two most used techniques in software process simulation modeling [15], it provides evidence of the feasibility of transformable process modeling in another dimension of the model transformation space.

8. CONCLUSIONS AND FUTURE WORK

As software systems have become more complex and network-centric than ever, the trustworthiness of a software product is becoming more critical and implies more properties over years.

The TPMF aims to deliver trustworthy software products. In order to support TPMF and leverage the trustworthy software development process, this paper provides the following contributions:

- A concept of trustworthy software process model is proposed to facilitate software process management and improvement, particularly as the inputs to TPMF.
- A new process methodology, transformable process modeling, is introduced to support the development of trustworthy software process models.
- We report an exploratory study of process model transformation between different modeling paradigms.

To be specific, this paper also make a number of technical contributions to software process modeling and model transformation through the exploratory study:

- At the structural semantics level, we identify a mapping from CLD and equations of SD to ASD of qualitative modeling, and vice versa.
- At the elementary semantics level, a model transformation scheme from a quantitative (SD) model to a qualitative (and semi-quantitative) model is implemented by using the semantic element mapping. From a given qualitative model and some additional quantification, it is possible to construct the corresponding SD model.
- At the composable semantics level, the n-th-order delay is introduced into semi-quantitative modeling and simulation, and implemented in QSIM algorithm framework.
- The software evolution processes are revisited by using qualitative and semi-quantitative modeling and simulation.
- We demonstrate that using transformable process modeling, quantitative software process (SD) models become comparable with the corresponding qualitative models.

However, as an exploratory study, this paper has not investigated transformations in other dimensions. Future research on transformable process modeling should include:

- Reinforcing the theoretical basis for transformable process modeling by integrating related research areas.
- Investigating process model transformabilities between other modeling paradigms or languages in more dimensions.
- Developing tools to facilitate automatic (or semi-automatic) transformation between process models in order to minimize effort.

ACKNOWLEDGEMENTS

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.
REFERENCES


**AUTHORS’ BIOGRAPHIES**

**He Zhang** is a researcher at National ICT Australia (NICTA) and a conjoint lecturer at the University of New South Wales (UNSW). Prior to joining NICTA, he was a research fellow at Lero, The Irish Software Engineering Research Centre. He joined academia after 7 years of experience in the industry, developing software systems in the areas of aerospace, manufacturing, and complex data management. He received his PhD in Software Engineering from UNSW, with a doctoral research that applied simulation modeling to study software engineering processes. He also holds an MPM from the University of Sydney, an MSc from the Chinese Academy of Astronautics, and a BEng from Nanjing University of Aeronautics and Astronautics in China. His research focuses on software process modeling, analysis and simulation, evidence-based software engineering, empirical software engineering, embedded systems development, service-oriented architecture and software project management. He is a member of ACM SIGSOFT.

**Barbara Kitchenham** is Professor of Quantitative Software Engineering at Keele University in the U.K. She has worked in software engineering for over 30 years both in the industry and in academia. Her main research interest is software measurement and its application to project management, quality control, risk management and evaluation of software technologies. Her most recent research has focused on the application of evidence-based practice to software engineering. She is a Chartered Mathematician and Fellow of the Institute of Mathematics and Its Applications, a Fellow of the Royal Statistical Society and a member of the IEEE Computer Society.
Ross Jeffery is Professor Emeritus of Software Engineering in the School of Computer Science and Engineering at UNSW and Research Leader in Empirical Software Engineering in the National ICT Australia (NICTA). Prior to his current positions he was the founding Head of the School of Information Systems at UNSW and Associate Dean (Technology) for the Faculty of Commerce and Economics. His research has involved over fifty government and industry organizations. He has also held positions at The University of Queensland, University of Maryland, and University of Minnesota. He serves on the editorial board of IEEE Transactions on Software Engineering and is the Associate Editor of the Journal of Empirical Software Engineering.